VISIT…

Evolutionary Intelligence

S. Sumathi · T. Hamsapriya · P. Surekha

Evolutionary Intelligence

An Introduction to Theory and Applications with Matlab

Springer

S. Sumathi
T. Hamsapriya
P. Surekha
PSG College of Technology
Coimbatore
India

ISBN: 978-3-540-75158-8 e-ISBN: 978-3-540-75382-7

Library of Congress Control Number: 2007938318

© 2008 Springer-Verlag Berlin Heidelberg

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilm or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

The use of general descriptive names, registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

Cover Design: Erich Kirchner, Heidelberg

Printed on acid-free paper

9 8 7 6 5 4 3 2 1

springer.com

Preface

EVOLUTIONARY INTELLIGENCE: Theoretical Advances and Applications in Evolutionary Algorithms, Genetic Algorithms, Genetic Programming and Parallel Genetic Algorithms is intended to help, provide basic information, and serve as a first straw for individuals, who are stranded in the mind boggling universe of Evolutionary Computation (EC). Over the course of the past years, global optimization algorithms imitating certain principles of nature have proved their usefulness in various domains of applications. Especially worth learning are those principles where nature has found "stable islands" in a "turbulent ocean" of solution possibilities. Such phenomena can be found in annealing processes, central nervous systems and biological Evolution, which in turn have lead to the following optimization methods: Simulated Annealing (SA), Artificial Neural Networks (ANNs) and the field of Evolutionary Computation (EC).

EC may currently be characterized by the following pathways: Genetic Algorithms (GA), Evolutionary Programming (EP), Evolution Strategies (ES), Classifier Systems (CFS), Genetic Programming (GP), and several other problem solving strategies, that are based upon biological observations, that date back to Charles Darwin's discoveries in the 19th century: the means of natural selection and the survival of the fittest, and theories of evolution. The inspired algorithms are thus termed Evolutionary Algorithms (EA).

Despite decades of work in evolutionary algorithms, there remains a lot of uncertainty as to when it is beneficial or detrimental to use recombination or mutation. This book provides a characterization of the roles that recombination and mutation play in evolutionary algorithms. Primary areas of coverage include the theory, implementation, and application of genetic algorithms (GAs), genetic programming (GP), parallel genetic algorithm (PGAs) and other variants of genetic and evolutionary computation. This book is ideal for researchers, engineers, computer scientists, graduate students who consider what frontiers await their exploration.

About the Book

This book gives a good introduction to evolutionary computation for those who are first entering the field and are looking for insight into the underlying mechanisms behind them. Emphasizing the scientific and machine learning applications of genetic algorithms instead of applications to optimization and engineering, the book could serve well in an actual course on adaptive algorithms. The author includes excellent problem sets, these being divided up into "thought exercises" and "computer exercises" in genetic algorithm. Practical use of genetic algorithms demands an understanding of how to implement them, and the author does so in the last two chapters of the book by giving the applications in various fields. This book also outlines some ideas on when genetic algorithms and genetic programming should be used, and this is useful since a newcomer to the field may be tempted to view a genetic algorithm as merely a fancy Monte Carlo simulation. The most difficult part of using a genetic algorithm is how to encode the population, and the author discusses various ways to do this. Various "exotic" approaches to improve the performance of genetic algorithms are also discussed such as the "messy" genetic algorithms, adaptive genetic algorithm and hybrid genetic algorithm.

Salient Features

The salient features of this book includes,

- Detailed description on Evolutionary Computation Concepts such as Evolutionary Algorithms, Genetic Algorithm and Genetic Programming
- A thorough understanding of Parallel Genetic Algorithm and its models
- Worked out examples using MATLAB software
- Application case studies based on MATLAB in various fields like Control Systems, Electronics, Image Processing, Power Electronics, Signal Processing, Communication and VLSI Design.
- MATLAB Toolboxes for Evolutionary Computation, Research Projects, Genetic Algorithm Codes in 'C' language, Abbreviations EC class/code libraries and software kits & Glossary.

Organization of the Book

Chapter 1 describes the history of evolutionary computation in brief and discusses the biological and artificial evolutionary process. The basic principle of evolution – Darwin's theory of evolution is described in detail in this chapter. A note on Genetics and the four important paradigms of EC are discussed in detail. An introduction to

global optimization and global optimization techniques such as Branch and Bound, Hybrid models, Tabu search etc., are elaborated in this chapter.

Chapter 2 provides an insight into the principles of evolutionary algorithms and its structure. This chapter also elaborates on the components of EAs as representation, fitness function, population initialization, selection, recombination, mutation, reinsertion and reproduction. EAs often take less time to find the optimal solution than gradient methods. However, most real-world problems involve simultaneous optimization of several often mutually concurrent objectives. Multi-objective EAs are able to find optimal trade-offs in order to get a set of solutions that are optimal in an overall sense. This chapter also discusses Multi-objective optimization using evolutionary algorithms.

Chapter 3 deals with the evolution of GA along with the operational functionalities of basic GA. The schema theorem is described with suitable illustrations. Some of the advanced operators that are used in GA are discussed and the types of GA such as parallel, sequential, hybrid, adaptive etc., are discussed in this chapter. The chapter also provides a set of basic MATLAB illustrations along with suitable codes.

In chapter 4, a brief history of genetic programming is discussed. To get an idea about programming a basic introduction to Lisp Programming Language is dealt. The basic operations of GP are discussed along with an illustration. The steps of GP are also illustrated along with a flow chart and enhanced versions of GP such as Meta GP, Cartesian GP and Strongly Typed GP are also elaborated in this chapter.

Chapter 5 deals with Parallel GA and an overview of parallel and distributed computer architecture. The classification of PGA and the various models are explained in detail with necessary illustrations. Communication topologies play a vital role in the classification of PGA. The advantages of PGA are also delineated.

Chapter 6 presents the applications of EA in the area of Biometric processing, engineering design, grammar development, waveform synthesis, scheduling earth observing satellites and portfolio optimization for general risk measures.

Chapter 7 presents the recent approaches of Genetic Algorithm with application examples in the areas of control systems, electronics, image processing, power electronics, signal processing, VLSI and Communication Systems.

Chapter 8 illustrates the application examples based on Genetic Programming in the areas of robotics, biomedical, physical science, interprocess communication, civil engineering, process control systems and artificial intelligence.

Chapter 9 describes the Parallel GA application in timetable problem solving, assembling DNA Fragments, solving Job Shop Scheduling Problems, Graph Coloring Problems and Ordering Problems.

About the Authors

The Author **Dr.S. Sumathi** born on 31.01.1968 completed B.E Degree in Electronics and Communication Engineering and Masters Degree in Applied Electronics at Government College of Technology, Coimbatore, TamilNadu. The Author got Ph.D.

Degree in the area of Data Mining currently, working as Assistant Professor in the Department of Electrical and Electronics Engineering, PSG College of Technology, Coimbatore with teaching and research experience of 16 years.

The Author received the prestigious Gold Medal from the Institution of Engineers Journal Computer Engineering Division - Subject Award for the research paper titled, "Development of New Soft Computing Models for Data Mining" 2002-2003 and also Best project award for UG Technical Report titled, "Self Organized Neural Network Schemes: As a Data mining tool", 1999. She received Dr.R Sundramoorthy award for Outstanding Academic of PSG College of Technology in the year 2006. The author has guided a project which received Best M.Tech Thesis award from Indian Society for Technical Education, New Delhi.

In appreciation of publishing various technical articles the Author has received National and International Journal Publication Awards – 2000–2003. She prepared Manuals for Electronics and Instrumentation Lab and Electrical and Electronics Lab of EEE Department, PSG College of Technology, Coimbatore and also organized second National Conference on Intelligent and Efficient Electrical Systems in the year 2005 and conducted Short Term Courses on "Neuro Fuzzy System Principles and Data Mining Applications", November 2001 & 2003. Dr.Sumathi has published several research articles in National & International Journals/Conferences and guided many UG and PG projects. She has also published books on, "Introduction to Neural Networks with MATLAB", "Introduction to Fuzzy Systems with MATLAB", "Introduction to Data mining and its Applications" and "LabVIEW based Advanced Instrumentation Systems" She reviewed papers in National / International Journals and Conferences. The Research interests include Neural Networks, Fuzzy Systems and Genetic Algorithms, Pattern Recognition and Classification, Data Warehousing and Data Mining, Operating systems and Parallel Computing etc.

Ms.T. Hamsapriya was born at Salem on 26th September 1967. She received her B.E and M.E degree from P.S.G College of Technology, Coimbatore. She has submitted her Ph.D thesis in Parallel Computing .She is currently working as an Assistant Professor in the Department of Computer Science and Engineering, PSG College of Technology, Coimbatore. She has four years of Industrial experience and twelve years of teaching experience.

She is a life member of ACS, SSI and ISTE. She is closely associated with many industrial training programmes. She has published several research papers in National and International Journals. She has reviewed papers and chaired many National/International Conferences. Her area of interests includes Parallel and Distributed Computing, Grid Computing, Advanced Computer Architecture, Genetic algorithms and Neural networks.

Ms. Surekha P has completed her B.E Degree in Electrical and Electronics Engineering in PARK College of Engineering and Technology, Coimbatore, Tamil Nadu, and Masters Degree in Control Systems at PSG College of Technology, Coimbatore, Tamil Nadu. She was a Rank Holder in both B.E and M.E Degree programme. She has received Alumni Award for best performance in curricular and co-curricular activities during her Masters Degree programme. She has presented papers in National

Conferences and Journals. She has also published a book “LabVIEW based Advanced Instrumentation Systems” She is presently working as a lecturer in Adhiyamaan College of Engineering, Hosur, Tamil Nadu. Her research areas include Robotics, Virtual Instrumentation, Mobile Communication, Neural Network, Fuzzy Logic Theory and Genetic Algorithm.

Acknowledgement

The authors are always thankful to the Almighty for perseverance and achievements.

The authors owe their gratitude to Mr. G Rangaswamy, Managing Trustee, PSG Institutions, and Dr. R.Rudramoorthy, Principal, PSG College of Technology, Coimbatore, for their whole-hearted cooperation and great encouragement given in this successful endeavour.

Dr.Sumathi owes much to her Daughter S. Priyanka, who has helped a lot in monopolizing her time on book work and substantially realized the responsibility. She feels happy and proud for the steel frame support rendered by her Husband.

Dr. Sumathi would like to extend whole-hearted thanks to her Parents who have reduced the family commitments and their constant support. She is greatly thankful to her Brother who has always been “Stimulator” for her progress. She is pertinent in thanking Parents-in-Laws for their great moral support.

Ms. Hamsapriya T would like to express her heart felt thanks to her husband and her daughter Shruthi for their untiring support and encouragement. Their unbounded love made this endeavour a reality. She is grateful to her father and father-in-law for their confidence in her which motivated in this work.

Ms. Surekha P would like to thank her parents, brother and husband who shouldered a lot of extra responsibilities during the months this was being written. They did this with the long term vision, depth of character, and positive outlook that are truly befitting of their name.

The authors wish to thank all their friends and colleagues who have been with them in all their endeavours with their excellent, unforgettable help and assistance in the successful execution of the work.

Contents

1 Introduction to Evolutionary Computation 1
1.1 Introduction 1
1.2 Brief History 2
1.3 Biological and Artificial Evolution 3
1.3.1 EC Terminology 4
1.3.2 Natural Evolution – The Inspiration from Biology 4
1.4 Darwinian Evolution 5
1.4.1 The Premise 6
1.4.2 Natural Selection 6
1.4.3 Slowly but Surely Process 7
1.4.4 A Theory in Crisis 7
1.4.5 Darwin's Theory of Evolution 8
1.5 Genetics 9
1.5.1 The Molecular Basis for Inheritance 9
1.6 Evolutionary Computation 10
1.7 Important Paradigms in Evolutionary Computation 12
1.7.1 Genetic Algorithms 12
1.7.2 Genetic Programming 14
1.7.3 Evolutionary Programming 15
1.7.4 Evolution Strategies 17
1.8 Global Optimization 20
1.9 Techniques of Global Optimization 21
1.9.1 Branch and Bound 21
1.9.2 Clustering Methods 22
1.9.3 Hybrid Methods 22
1.9.4 Simulated Annealing 26
1.9.5 Statistical Global Optimization Algorithms 27
1.9.6 Tabu Search 27
1.9.7 Multi Objective Optimization 28
Summary 30
Review Questions 30

2 Principles of Evolutionary Algorithms 31
2.1 Introduction 31
2.2 Structure of Evolutionary Algorithms 32
2.2.1 Illustration 34
2.3 Components of Evolutionary Algorithms 36
2.4 Representation 36
2.5 Evaluation/Fitness Function 37
2.6 Population Initialization 37
2.7 Selection 38
2.7.1 Rank Based Fitness Assignment 39
2.7.2 Multi–objective Ranking 41
2.7.3 Roulette Wheel selection 43
2.7.4 Stochastic universal sampling 44
2.7.5 Local Selection 45
2.7.6 Truncation Selection 47
2.7.7 Comparison of Selection Properties 49
2.7.8 MATLAB Code Snippet for Selection 51
2.8 Recombination 52
2.8.1 Discrete Recombination 52
2.8.2 Real Valued Recombination 53
2.8.3 Binary Valued Recombination (Crossover) 57
2.9 Mutation 61
2.9.1 Real Valued Mutation 62
2.9.2 Binary mutation 63
2.9.3 Real Valued Mutation with Adaptation of Step Sizes 64
2.9.4 Advanced Mutation 65
2.9.5 Other Types of Mutation 66
2.9.6 MATLAB Code Snippet for Mutation 67
2.10 Reinsertion 67
2.10.1 Global Reinsertion 67
2.10.2 Local Reinsertion 68
2.11 Reproduction Operator 69
2.11.1 MATLAB Code Snippet for Reproduction 70
2.12 Categorization of Parallel Evolutionary Algorithms 70
2.13 Advantages of Evolutionary Algorithms 72
2.14 Multi-objective Evolutionary Algorithms 73
2.15 Critical Issues in Designing an Evolutionary Algorithm 74
Summary 75
Review Questions 75

3 Genetic Algorithms with Matlab 77
3.1 Introduction 77
3.2 History of Genetic Algorithm 80
3.3 Genetic Algorithm Definition 81
3.4 Models of Evolution 82

3.5 Operational Functionality of Genetic Algorithms 83
3.6 Genetic Algorithms – An Example 84
3.7 Genetic Representation 86
3.8 Genetic Algorithm Parameters 86
3.8.1 Multi-Parameters 86
3.8.2 Concatenated, Multi-Parameter, Mapped, Fixed-Point Coding 87
3.8.3 Exploitable Techniques 87
3.9 Schema Theorem and Theoretical Background 88
3.9.1 Building Block Hypothesis 89
3.9.2 Working of Genetic Algorithms 90
3.9.3 Sets and Subsets 91
3.9.4 The Dynamics of a Schema 92
3.9.5 Compensating for Destructive Effects 93
3.9.6 Mathematical Models 94
3.9.7 Illustrations based on Schema Theorem 97
3.10 Solving a Problem: Genotype and Fitness 100
3.10.1 Non-Conventional Genotypes 102
3.11 Advanced Operators in GA 104
3.11.1 Inversion and Reordering 104
3.11.2 Epistasis 104
3.11.3 Deception 104
3.11.4 Mutation and Naive Evolution 105
3.11.5 Niche and Speciation 105
3.11.6 Restricted Mating 105
3.11.7 Diploidy and Dominance 106
3.12 Important Issues in the Implementation of a GA 106
3.13 Comparison of GA with Other Methods 107
3.13.1 Neural Nets 107
3.13.2 Random Search 107
3.13.3 Gradient Methods 107
3.13.4 Iterated Search 108
3.13.5 Simulated Annealing 108
3.14 Types of Genetic Algorithm 109
3.14.1 Sequential GA 109
3.14.2 Parallel GA 110
3.14.3 Hybrid GA 112
3.14.4 Adaptive GA 113
3.14.5 Integrated Adaptive GA (IAGA) 116
3.14.6 Messy GA 117
3.14.7 Generational GA (GGA) 120
3.14.8 Steady State GA (SSGA) 121
3.15 Advantages of GA 122
3.16 Matlab Examples of Genetic Algorithms 123
3.16.1 Genetic Algorithm Operations Implemented in MATLAB . . 123
3.16.2 Illustration 1 – Maximizing the given Function 126

3.16.3 Illustration 2 – Optimization of a Multidimensional Non-Convex Function . . . 132
3.16.4 Illustration 3 – Traveling Salesman Problem . . . 136
3.16.5 Illustration 4 – GA using Float Representation . . . 143
3.16.6 Illustration 5 – Constrained Problem . . . 158
3.16.7 Illustration 6 – Maximum of any given Function . . . 161
Summary . . . 167
Review Questions . . . 169

4 Genetic Programming Concepts . . . 171
4.1 Introduction . . . 171
4.2 A Brief History of Genetic Programming . . . 175
4.3 The Lisp Programming Language . . . 176
4.4 Operations of Genetic Programming . . . 177
4.4.1 Creating an Individual . . . 177
4.4.2 Creating a Random Population . . . 178
4.4.3 Fitness Test . . . 179
4.4.4 Functions and Terminals . . . 179
4.4.5 The Genetic Operations . . . 179
4.4.6 Selection Functions . . . 180
4.4.7 Crossover Operation . . . 182
4.4.8 Mutation . . . 183
4.4.9 User Decisions . . . 183
4.5 An Illustration . . . 185
4.6 The GP Paradigm in Machine Learning . . . 186
4.7 Preparatory Steps of Genetic Programming . . . 188
4.7.1 The Terminal Set . . . 188
4.7.2 The Function Set . . . 189
4.7.3 The Fitness Function . . . 189
4.7.4 The Algorithm Control Parameters . . . 189
4.7.5 The Termination Criterion . . . 190
4.8 Flow – Chart of Genetic Programming . . . 191
4.9 Type Constraints in Genetic Programming . . . 193
4.10 Enhanced Versions of Genetic Programming . . . 195
4.10.1 Meta-genetic Programming . . . 196
4.10.2 Cartesian Genetic Programming . . . 201
4.10.3 Strongly Typed Genetic Programming (STGP) . . . 209
4.11 Advantages of using Genetic Programming . . . 217
Summary . . . 217
Review Questions . . . 218

5 Parallel Genetic Algorithms . . . 219
5.1 Introduction . . . 219
5.2 Parallel and Distributed Computer Architectures: An Overview . . . 220
5.3 Classification of PGA . . . 223

5.4 Parallel Population Models for Genetic Algorithms 224
5.4.1 Classification of Global Population Models 225
5.4.2 Global Population Models 226
5.4.3 Regional Population Models 226
5.4.4 Local Population Models 228
5.5 Models Based on Distribution of Population 230
5.5.1 Centralized PGA 230
5.5.2 Distributed PGA 231
5.6 PGA Models Based on Implementation 232
5.6.1 Master–slave/Farming PGA 232
5.6.2 Island PGA 234
5.6.3 Cellular PGA 236
5.7 PGA Models Based on Parallelism 238
5.7.1 Global with Migration (coarse-grained) 238
5.7.2 Global with Migration (fine-grained) 238
5.8 Communication Topologies 240
5.9 Hierarchical Parallel Algorithms 241
5.10 Object Orientation (OO) and Parallelization 243
5.11 Recent Advancements 244
5.12 Advantages of Parallel Genetic Algorithms 246
Summary 247
Review Questions 247

6 Applications of Evolutionary Algorithms 249
6.1 A Fingerprint Recognizer using Fuzzy Evolutionary Programming . 249
6.1.1 Introduction 249
6.1.2 Fingerprint Characteristics 250
6.1.3 Fingerprint Recognition using EA 255
6.1.4 Experimental Results 257
6.1.5 Conclusion and Future Work 258
6.2 An Evolutionary Programming Algorithm for Automatic Engineering Design 258
6.2.1 Introduction 258
6.2.2 EPSOC: An Evolutionary Programming Algorithm using Self-Organized Criticality 260
6.2.3 Case Studies 261
6.2.4 Results of Numerical Experiments 263
6.2.5 Conclusion 265
6.3 Evolutionary Computing as a Tool for Grammar Development 265
6.3.1 Introduction 265
6.3.2 Natural Language Grammar Development 266
6.3.3 Grammar Evolution 267
6.3.4 GRAEL-1: Probabilistic Grammar Optimization 268
6.3.5 GRAEL-2: Grammar Rule Discovery 272

6.3.6 GRAEL-3: Unsupervised Grammar Induction 274
6.3.7 Concluding Remarks 275
6.4 Waveform Synthesis using Evolutionary Computation 276
6.4.1 Introduction 276
6.4.2 Evolutionary Manipulation of Waveforms 276
6.4.3 Conclusion and Results 279
6.5 Scheduling Earth Observing Satellites with Evolutionary Algorithms 282
6.5.1 Introduction 282
6.5.2 EOS Scheduling by Evolutionary Algorithms and other Optimization Techniques 284
6.5.3 Results 286
6.5.4 Future Work 288
6.6 An Evolutionary Computation Approach to Scenario-Based Risk-return Portfolio Optimization 289
6.6.1 Introduction 289
6.6.2 Portfolio Optimization 289
6.6.3 Evolutionary Portfolio Optimization 291
6.6.4 Numerical Results 292
6.6.5 Results 293
6.6.6 Conclusion 296

7 Applications of Genetic Algorithms 297
7.1 Assembly and Disassembly Planning by Using Fuzzy Logic & Genetic Algorithms 297
7.1.1 Research Background 298
7.1.2 Proposed Approach and Case Studies 303
7.1.3 Discussion of Results 305
7.1.4 Concluding Remarks 308
7.2 Automatic Synthesis of Active Electronic Networks Using Genetic Algorithms 308
7.2.1 Active Network Synthesis Using GAs 309
7.2.2 Example of an Automatically-Synthesized Network 311
7.2.3 Limitations of Automatic Network Synthesis 313
7.2.4 Concluding Remarks 313
7.3 A Genetic Algorithm for Mixed Macro and Standard Cell Placement 314
7.3.1 Genetic Algorithm for Placement 314
7.3.2 Experimental Results 318
7.4 Knowledge Acquisition on Image Procssing Based on Genetic Algorithms 319
7.4.1 Methods 320
7.4.2 Results and Discussions 325
7.4.3 Concluding Remarks 327

7.5 Map Segmentation by Colour Cube Genetic *K*-Mean Clustering ... 327
7.5.1 Genetic Clustering in Image Segmentation ... 328
7.5.2 K-Means Clustering Model ... 329
7.5.3 Genetic Implementation ... 329
7.5.4 Results and Conclusions ... 330
7.6 Genetic Algorithm-Based Performance Analysis of Self-Excited Induction Generator ... 331
7.6.1 Modelling of SEIG System ... 332
7.6.2 Genetic Algorithm Optimization ... 334
7.6.3 Results and Discussion ... 335
7.6.4 Concluding Remarks ... 337
7.7 Feature Selection for Anns Using Genetic Algorithms in Condition Monitoring ... 338
7.7.1 Signal Acquisition ... 340
7.7.2 Neural Networks ... 340
7.7.3 Genetic Algorithms ... 341
7.7.4 Training and Simulation ... 341
7.7.5 Results ... 342
7.7.6 Concluding Remarks ... 343
7.8 Parallel Space-Time Adaptive Processing Algorithms ... 343
7.8.1 Overview of Parallel STAP ... 344
7.8.2 Genetic Algorithm Methodology ... 345
7.8.3 Numerical Results ... 347
7.8.4 Concluding Remarks ... 348
7.9 A Multi-Objective Genetic Algorithm for on-Chip Real-Time Adaptation of a Multi-Carrier Based Telecommunications Receiver ... 349
7.9.1 MC-CDMA Receiver ... 350
7.9.2 Multi-objective Genetic Algorithm (GA) ... 350
7.9.3 Results ... 353
7.9.4 Concluding Remarks ... 355
7.10 A VLSI Implementation of an Analog Neural Network Suited for Genetic Algorithms ... 355
7.10.1 Realization of the Neural Network ... 357
7.10.2 Implementation of the Genetic Training Algorithm ... 362
7.10.3 Experimental Results ... 364
7.10.4 Concluding Remarks ... 366

8 Genetic Programming Applications ... 367
8.1 GP-Robocode: Using Genetic Programming to Evolve Robocode Players ... 367
8.1.1 Robocode Rules ... 368
8.1.2 Evolving Robocode Strategies using Genetic Programming ... 369
8.1.3 Results ... 374
8.1.4 Concluding Remarks ... 375

8.2 Prediction of Biochemical Reactions using Genetic Programming . . 375
8.2.1 Method and Results 376
8.2.2 Discussion 377
8.3 Application of Genetic Programming to High Energy Physics Event Selection 377
8.3.1 Genetic Programming 378
8.3.2 Combining Genetic Programming with High Energy Physics Data 380
8.3.3 Selecting Genetic Programming Parameters 385
8.3.4 Testing Genetic Programming on $D^+ \rightarrow K^+\pi^+\pi^-$ 389
8.3.5 Concluding Remarks 394
8.4 Using Genetic Programming to Generate Protocol Adaptors for Interprocess Communication 395
8.4.1 Prerequisites of Interprocess Communication 397
8.4.2 Specifying Protocols 397
8.4.3 Evolving Protocols 400
8.4.4 The Experiment 403
8.4.5 Concluding Remarks 405
8.5 Improving Technical Analysis Predictions: An Application of Genetic Programming 406
8.5.1 Background 407
8.5.2 FGP for Predication in DJIA Index 408
8.5.3 Concluding Remarks 411
8.6 Genetic Programming within Civil Engineering 412
8.6.1 Generational Genetic Programming 412
8.6.2 Applications of Genetic Programming in Civil Engineering . 413
8.6.3 Application of Genetic Programming in Structural Engineering 413
8.6.4 Structural Encoding 413
8.6.5 An Example of Structural Optimization 414
8.6.6 10 Member Planar Truss 415
8.6.7 Controller-GP Tableau 415
8.6.8 Model 416
8.6.9 View-Visualisation 417
8.6.10 Concluding Remarks 419
8.7 Chemical Process Controller Design using Genetic Programming . . 420
8.7.1 Dynamic Reference Control Problem 420
8.7.2 ARX Process Description 423
8.7.3 CSTR (Continuous Stirred Tank Reactor) Process Description 423
8.7.4 GP Problem Formulation 425
8.7.5 GP Configuration and Implementation Aspects 426
8.7.6 Results 428
8.7.7 Concluding Remarks 430

8.8 Trading Applications of Genetic Programming 431
8.8.1 Application: Forecasting or Prediction 433
8.8.2 Application: Finding Causal Relationships 434
8.8.3 Application: Building Trading Rules 434
8.8.4 Concluding Remarks 435
8.9 Artificial Neural Network Development by Means of Genetic Programming with Graph Codification 435
8.9.1 State of the Art 435
8.9.2 Model 438
8.9.3 Problems to be Solved 441
8.9.4 Results and Comparison with Other Methods 441
8.9.5 Concluding Remarks 443

9 Applications of Parallel Genetic Algorithm 445
9.1 Timetabling Problem 445
9.1.1 Introduction 445
9.1.2 Applying Genetic Algorithms to Timetabling 446
9.1.3 A Parallel Algorithm 450
9.1.4 Results 452
9.1.5 Conclusion 453
9.2 Assembling DNA Fragments with a Distributed Genetic Algorithm . 453
9.2.1 Introduction 453
9.2.2 The DNA Fragment Assembly Problem 454
9.2.3 DNA Sequencing Process 455
9.2.4 DNA Fragment Assembly Using the Sequential GA 457
9.2.5 Implementation Details 458
9.2.6 DNA Fragment Assembly Problem using the Parallel GA . . 460
9.2.7 Experimental Results 462
9.3 Investigating Parallel Genetic Algorithms on Job Shop Scheduling Problems 469
9.3.1 Introduction 469
9.3.2 Job Shop Scheduling Problem 470
9.3.3 Genetic Representation and Specific Operators 471
9.3.4 Parallel Genetic Algorithms for JSSP 473
9.3.5 Computational Results 475
9.3.6 Comparison of PGA Models 477
9.4 Parallel Genetic Algorithm for Graph Coloring Problem 479
9.4.1 Introduction 479
9.4.2 Genetic Operators for GCP 480
9.4.3 Experimental Verification 484
9.4.4 Conclusion 486
9.5 Robust and Distributed Genetic Algorithm for Ordering Problems 486
9.5.1 Introduction 486
9.5.2 Ordering Problems 487

9.5.3 Traveling Salesman Problem 488
9.5.4 Distributed Genetic Algorithm 491

Appendix – A Glossary 503

Appendix – B Abbreviations 517

Appendix – C Research Projects 521
C.1 Evolutionary Simulation-based Validation 521
C.2 Automatic Generation of Validation Stimuli for Application-specific Processors 521
C.3 Dynamic Prediction of Web Requests 522
C.4 Analog Genetic Encoding for the Evolution of Circuits and Networks 522
C.5 An Evolutionary Algorithm for Global Optimization Based on Level-set Evolution and Latin Squares 522
C.6 Imperfect Evolutionary Systems 523
C.7 A Runtime Analysis of Evolutionary Algorithms for Constrained Optimization Problems 523
C.8 Classification with Ant Colony Optimization 524
C.9 Multiple Choices and Reputation in Multiagent Interactions 524
C.10 Coarse-grained Dynamics for Generalized Recombination 525
C.11 An Evolutionary Algorithm-based Approach to Automated Design of Analog and RF Circuits Using Adaptive Normalized Cost Functions 525
C.12 An Investigation on Noisy Environments in Evolutionary Multi-objective Optimization 526
C.13 Interactive Evolutionary Computation-based Hearing Aid Fitting . . . 527
C.14 Evolutionary Development of Hierarchical Learning Structures 527
C.15 Knowledge Interaction with Genetic Programming in Mechatronic Systems Design Using Bond Graphs 528
C.16 A Distributed Evolutionary Classifier for Knowledge Discovery in Data Mining 528
C.17 Evolutionary Feature Synthesis for Object Recognition 528
C.18 Accelerating Evolutionary Algorithms with Gaussian Process Fitness Function Models 529
C.19 A Constraint-based Genetic Algorithm Approach for Mining Classification Rules 529
C.20 An Evolutionary Algorithm for Solving Nonlinear Bilevel Programming Based on a New Constraint-handling Scheme 530
C.21 Evolutionary Fuzzy Neural Networks for Hybrid Financial Prediction 530
C.22 Genetic Recurrent Fuzzy System by Coevolutionary Computation with Divide-and-Conquer Technique 531
C.23 Knowledge-based Fast Evaluation for Evolutionary Learning 531

C.24 A Comparative Study of Three Evolutionary Algorithms Incorporating Different Amounts of Domain Knowledge for Node Covering Problem 532

Appendix – D MATLAB Toolboxes 533

Appendix – E Commercial Software Packages 537

Appendix – F Ga Source Codes in 'C' Language 547

Appendix – G EC Class/Code Libraries and Software Kits 559

Bibliography 569

Chapter 1
Introduction to Evolutionary Computation

Learning Objectives: On completion of this chapter the reader will have knowledge on:

- History of Evolutionary computation
- The inspiration of Natural Evolution from Biology
- Darwin's theory of evolution
- The molecular basis for inheritance
- Paradigms in Evolutionary Computation such as Genetic Algorithms Genetic Programming, Evolutionary Programming, Evolution Strategies
- Techniques of Global Optimization such as Branch and Bound, Clustering Methods, Hybrid Methods, Simulated Annealing, Statistical Global Optimization Algorithms, Tabu Search, Multi objective optimization

1.1 Introduction

An important area in current research is the development and application of search techniques based upon the principles of natural evolution. Most readers, through the popular literature and typical Western educational experience, are probably aware of the basic concepts of evolution. In particular, the principle of the '*survival of the fittest*' proposed by Charles Darwin (1859) has especially captured the popular imagination. The theory of natural selection proposes that the plants and animals that exist today are the result of millions of years of adaptation to the demands of the environment. At any given time, a number of different organisms may co-exist and compete for the same resources in an ecosystem. The organisms that are most capable of acquiring resources and thus procreating are the ones whose descendants will tend to be numerous in the future. Organisms that are less capable, for whatever reason, will tend to have few or no descendants in the future. The former are said to be more *fit* than the latter, and the distinguishing characteristics that caused the former to be most fit are said to be *selected for* over the characteristics of the

latter. Over time, the entire population of the ecosystem is said to *evolve* to contain organisms that, on average, are more fit than those of previous generations of the population because they exhibit more of those characteristics that tend to promote survival.

Evolutionary computation techniques abstract these evolutionary principles into algorithms that may be used to search for optimal solutions to a problem. In a search algorithm, a number of possible solutions to a problem are available and the task is to find the best solution possible in a fixed amount of time. For a search space with only a small number of possible solutions, all the solutions can be examined in a reasonable amount of time and the optimal one found. This *exhaustive search*, however, quickly becomes impractical as the search space grows in size. Traditional search algorithms randomly sample (e.g., *random walk*) or heuristically sample (e.g., *gradient descent*) the search space one solution at a time in the hopes of finding the optimal solution. The key aspect distinguishing an evolutionary search algorithm from such traditional algorithms is that it is *population-based*. Through the adaptation of successive generations of a large number of individuals, an evolutionary algorithm performs an efficient directed search. Evolutionary search is generally better than random search and is not susceptible to the hill-climbing behaviors of gradient based search.

This chapter describes the history of evolutionary computation in brief and discusses the biological and artificial evolutionary process. The basic principle of evolution – Darwin's theory of evolution is described in detail in this chapter. A note on Genetics and the four important paradigms of EC are discussed in detail. An introduction to global optimization and global optimization techniques such as Branch and Bound, Hybrid models, Tabu search etc., are elaborated in this chapter.

1.2 Brief History

Natural evolution is a historical process: "Anything that changes in time, by definition, has a history". The essence of evolution is change. Evolution is a dynamic, two-step process of random variation and selection that engenders constant change to individuals within a population in response to environmental dynamics and demands. Individuals who are poorly adapted to their current environment are culled from the population. The adaptations that allow specific individuals to survive are heritable, coded in a complex genetic milieu that is passed on to successive progeny. Thus, adaptations are a historical sequence of consecutive events, each one leading to the next. The converse, however, is also true: History itself is an evolutionary process, not just in the sense that it changes over time but also in the more strict sense that it undergoes mutation and selection. An initial message, perhaps provided by a teacher, is sent from one child to another, whispered in the ear, then passed along again in sequence. Each child can only hear what the last child whispers. The message is subject to mutation at each step, occasionally by accident, sometimes with

devious intent. At last, the final student divulges the message that he or she heard, as it was understood. This "mutant" message is then compared to the actual message, revealed by the teacher. In effect, the teacher applies a form of selection to restore the original message. So it is with history: "Written histories, like science itself, are constantly in need of revision. Erroneous interpretations of an earlier author eventually become myths, accepted without question and carried forward from generation to generation".

The aim is to perform the function of selection on what has mostly been a process of mutation. Such error correction is surely needed for the conventional accounting of the history of evolutionary computation is more of a fable or science fiction than a factual record. Readers only casually acquainted with the field may be surprised to learn, for example, that the idea to use recombination in population-based evolutionary simulations did not arise in a single major innovation but in fact was commonly, if not routinely, applied in multiple independent lines of investigation in the 1950s and 1960s. It may also be of interest that the prospects of using computers to study "life-as-it-could-be" (i.e., artificial life) rather than "life-as-we-know-it" were plainly clear to population geneticists and evolutionary biologists in the same time frame.

In the fifties, long before computers were used on a great scale, the idea to use Darwinian principles for automated problem solving originated. In the sixties, three different interpretations of this idea have been developed at three different places. Evolutionary programming was introduced by Lawrence J. Fogel in the USA, while John Henry Holland called his method a genetic algorithm. In Germany Ingo Rechenberg and Hans-Paul Schwefel introduced evolution strategies. These areas developed separately for about 15 years. From the early nineties they are seen as different representatives ("dialects") of one technology, called evolutionary computing. In the early nineties, another fourth stream following the general ideas had emerged – genetic programming. These terminologies denote the whole field by evolutionary computing and consider evolutionary programming, evolution strategies, genetic algorithms, and genetic programming as sub-areas.

1.3 Biological and Artificial Evolution

Evolutionary Computation (EC) is based on the principles of biological evolution. In order to explain the formulation of each of the three different forms of EC, this section gives a basic introduction to some of the terminology used by the EC community and a brief overview of the key concepts of natural evolution upon which EAs are based.

1.3.1 EC Terminology

Table 1.1 presents a brief overview of some of the terminology borrowed from biology and used in EC.

1.3.2 Natural Evolution – The Inspiration from Biology

Evolution is the process of gradual development. Living organisms evolve through the interaction of competition, selection, reproduction and mutation processes. The evolution of a population of organisms highlights the differences between an organism's "genotype" and "phenotype." The genotype is the organism's underlying genetic coding (DNA). The phenotype is the manner of response contained in the physiology, morphology and behavior of the organism.

Although the basic principles of evolution (natural selection and mutation) are understood in the large, both population genetics and phylogenetics have been radically transformed by the recent availability of large quantities of molecular data. For example, in population genetics (study of mutations in populations), more molecular variability was found in the 1960s than had been expected. Phylogenetics (study of evolutionary history of life) makes use of a variety of different kinds of data, of which DNA sequences are the most important, as well as whole-genome, metabolic, morphological, geographical and geological data.

Evolutionary biology is found on the concept that organisms share a common origin and have diverged through time. The details and timing of these divergences-that is, the estimation or reconstruction of an evolutionary history are important for both intellectual and practical reasons, and phylogenies are central to virtually all comparisons among species. From a practical standpoint, phylogenetics has helped to trace routes to infectious disease transmission and to identify new pathogens.

There are many interesting phylogenetic problems. For example, consider the problem of estimating large phylogenesis, which is a central challenge in evolutionary

Table 1.1 A summary of the basic terminology used within EC

Biological term	EC meaning
Chromosome	String of symbols
Population	A set of chromosomes
Deme	A local population of closely related chromosomes, a subset of the total population
Gene	A feature, character or detector
Allele	Feature value
Locus	A position in a chromosome
Genotype	Structure
Phenotype	A set of parameters, an alternative solution or a decoded structure

biology. Given three species, there are only three possible trees that could represent their phylogenetic history: (A, (B, C)); (B, (A, C)); and (C, (A, B)). The notation (A, (B, C)) means that B and C share a common ancestor, who itself shares a different common ancestor with A. Thus even if one picks a tree in random, there is a one in three chance that the tree chosen will be correct. But the number of possible trees grows very rapidly with the number of species involved. For a small phylogenetic problem involving 10 species, there are 34,459,425 possible trees. For a problem involving 22 species, the number of trees exceeds 10^{23}. Today, most phylogenetic problems involve more than 80 species and some datasets contain more than 500 species.

Given the existence such large search spaces, it is clear that exhaustive search for the single correct phylogenetic tree is mot a feasible strategy, regardless of how fast computers become in the foreseeable future. Researchers have developed a number of methods for coping with the size of these problems, but many of these methods have serious deficiencies. Thus, the algorithmic of evolutionary biology is a fertile area for research.

A related problem is that of comparing one or more features across species. The comparative method has provided much of the evidence for natural selection and is probably the most widely used statistical method in evolutionary biology. But comparative analysis must account for phylogenetic history, since the similarity in features common to multiple species that originate in a common evolutionary history can inappropriately and seriously bias the analyses. A number of methods have been developed to accommodate phylogenics in comparative analysis, but most of these methods assume that the phylogeny is known without error. However, this is patently unrealistic, because almost all phylogenetics have a large degree of uncertainity. An important question is therefore to understand how comparative analysis can be performed that accommodate phylogenetic history without depending on any single phylogeny being correct.

Still another interesting problem concerns the genetics of adaptation – the genomic changes that occur when an organism adapts to a new set of selection pressure in a new environment. Because the process of adaptive change is difficult to study directly, there are many important and unanswered questions regarding the genetics of adaptation. For example, how many mutations are involved in a given adaptive change? Does this figure change when different organisms or different environments are involved? What is the distribution of fitness effects implied by these genetic changes during adaptation? Though some of the questions remain unanswered evolutionary trends are leading in almost all the research areas.

1.4 Darwinian Evolution

Darwin's "Origin of Species by Means of Natural Selection", or the "Preservation of Favored Races in the Struggle for Life", made several points that had major impact on nineteenth-century thought:

- That biological types or species do not have a fixed, static existence but exist in permanent states of change and flux;
- That all life, biologically considered, takes the form of a struggle to exist – more exactly, to exist and produce the greatest number of offspring;
- That this struggle for existence culls out those organisms less well adapted to any particular ecology and allows those better adapted to flourish – a process called Natural Selection;
- That natural selection, development, and evolution requires enormously long periods of time, so long, in fact, that the everyday experience of human beings provides them with no ability to interpret such histories;
- That the genetic variations ultimately producing increased survivability are random and not caused (as religious thinkers would have it) by God or (as Lammarck would have it) by the organism's own striving for perfection.

1.4.1 The Premise

Darwin's Theory of Evolution is the widely held notion that all life is related and has descended from a common ancestor: the birds and the bananas, the fishes and the flowers – all related. Darwin's general theory presumes the development of life from non-life and stresses a purely naturalistic (undirected) "descent with modification". That is, complex creatures evolve from more simplistic ancestors naturally over time. In a nutshell, as random genetic mutations occur within an organism's genetic code, the beneficial mutations are preserved because they aid survival – a process known as "natural selection." These beneficial mutations are passed on to the next generation. Over time, beneficial mutations accumulate and the result is an entirely different organism (not just a variation of the original, but an entirely different creature).

1.4.2 Natural Selection

While Darwin's Theory of Evolution is a relatively young archetype, the evolutionary worldview itself is as old as antiquity. Ancient Greek philosophers such as Anaximander postulated the development of life from non-life and the evolutionary descent of man from animal. Charles Darwin simply brought something new to the old philosophy – a plausible mechanism called "natural selection." Natural selection acts to preserve and accumulate minor advantageous genetic mutations. Suppose a member of a species developed a functional advantage (it grew wings and learned to fly). Its offspring would inherit that advantage and pass it on to their offspring. The inferior (disadvantage) members of the same species would gradually die out, leaving only the superior (advantage) members of the species. Natural selection is the preservation of a functional advantage that enables a species to compete better in

the wild. Natural selection is the naturalistic equivalent to domestic breeding. Over the centuries, human breeders have produced dramatic changes in domestic animal populations by selecting individuals to breed. Breeders eliminate undesirable traits gradually over time. Similarly, natural selection eliminates inferior species gradually over time.

1.4.3 Slowly but Surely Process

Darwin's Theory of Evolution is a slow gradual process. Darwin wrote, "Natural selection acts only by taking advantage of slight successive variations; it can never take a great and sudden leap, but must advance by short and sure, though slow steps." Thus, Darwin conceded that, "If it could be demonstrated that any complex organ existed, which could not possibly have been formed by numerous, successive, slight modifications, my theory would absolutely break down." Such a complex organ would be known as an "irreducibly complex system". An irreducibly complex system is one composed of multiple parts, all of which are necessary for the system to function. If even one part is missing, the entire system will fail to function. Every individual part is integral. Thus, such a system could not have evolved slowly, piece by piece. The common mousetrap is an everyday non-biological example of irreducible complexity. It is composed of five basic parts: a catch (to hold the bait), a powerful spring, a thin rod called "the hammer," a holding bar to secure the hammer in place, and a platform to mount the trap. If any one of these parts is missing, the mechanism will not work. Each individual part is integral. The mousetrap is irreducibly complex.

1.4.4 A Theory in Crisis

Darwin's Theory of Evolution is a theory in crisis in light of the tremendous advances made in molecular biology, biochemistry and genetics over the past fifty years. There are in fact tens of thousands of irreducibly complex systems on the cellular level. Specified complexity pervades the microscopic biological world. Molecular biologist Michael Denton wrote, "Although the tiniest bacterial cells are incredibly small, weighing less than 10^{-12} grams, each is in effect a veritable microminiaturized factory containing thousands of exquisitely designed pieces of intricate molecular machinery, made up altogether of one hundred thousand million atoms, far more complicated than any machinery built by man and absolutely without parallel in the non-living world."

There is no need for a microscope to observe irreducible complexity. The eye, the ear and the heart are all examples of irreducible complexity, though they were not recognized as such in Darwin's day. Nevertheless, Darwin confessed, "To suppose that the eye with all its inimitable contrivances for adjusting the focus to

different distances, for admitting different amounts of light, and for the correction of spherical and chromatic aberration, could have been formed by natural selection, seems, absurd in the highest degree."

1.4.5 Darwin's Theory of Evolution

Darwin's theory of evolution is based on five key observations and inferences drawn from them. These observations and inferences have been summarized by the great biologist Ernst Mayr as follows:

- First, species have great fertility. They make more offspring than can grow to adulthood.
- Second, populations remain roughly the same size, with modest fluctuations.
- Third, food resources are limited, but are relatively constant most of the time. From these three observations it may be inferred that in such an environment there will be a struggle for survival among individuals.
- Fourth, in sexually reproducing species, generally no two individuals are identical. Variation is rampant.
- Fifth, much of this variation is heritable.

From this it may be inferred that in a world of stable populations where each individual must struggle to survive, those with the "best" characteristics will be more likely to survive, and those desirable traits will be passed to their offspring. These advantageous characteristics are inherited by following generations, becoming dominant among the population through time. This is *natural selection*. It may be further inferred that natural selection, if carried far enough, makes changes in a population, eventually leading to new species. These observations have been amply demonstrated in biology, and even fossils demonstrate the veracity of these observations.

To summarise Darwin's Theory of Evolution;

1. Variation: There is Variation in Every Population.
2. Competition: Organisms Compete for limited resources.
3. Offspring: Organisms produce more Offspring than can survive.
4. Genetics: Organisms pass Genetic traits on to their offspring.
5. Natural Selection: Those organisms with the Most Beneficial Traits are more likely to survive and reproduce.

Darwin imagined it might be possible that all life is descended from an original species from ancient times. DNA evidence supports this idea. Probably all organic beings which have ever lived on this earth have descended from some one primordial form, into which life was first breathed. There is grandeur in this view of life that, whilst this planet has gone cycling on according to the fixed law of gravity, from so simple a beginning endless forms most beautiful and most wonderful have been, and are being evolved.

1.5 Genetics

Genetics (Greek: genno; to give birth) is the science of heredity and variation in living organisms. Knowledge that desired characteristics were inherited has been implicitly used since prehistoric times for improving crop plants and animals through selective breeding. However, the modern science of genetics, which seeks to understand the mechanisms of inheritance, only began with the work of Gregor Mendel in the mid-1800s. The genetic information of cellular organisms is contained within the chemical structure of DNA (deoxyribonucleic acid) molecules. Individually inherited traits, corresponding to regions in the DNA sequence, are called genes. Genes encode the information necessary for synthesizing proteins — complex molecules generally responsible for enzymatic reactions, synthesis, communication and structure within a cell. DNA sequence is transcribed into an intermediate RNA (ribonucleic acid) molecule, "messenger RNA", and ribosomes translate this sequence to form a chain of amino acids, thereby creating a protein molecule. It is through their proteins that most genes have a biological effect. Although genetics plays a large role in determining the appearance and behavior of organisms, it is the interaction of genetics with the environment that determines the ultimate outcome.

1.5.1 The Molecular Basis for Inheritance

The molecular basis for genes is the chemical deoxyribonucleic acid (DNA). DNA is composed of a chain of nucleotides, of which there are four types: adenine (A), cytosine (C), guanine (G), and thymine (T). Genetic information exists in the sequence of these nucleotides, and genes exist as stretches of sequence along the DNA chain. The molecular structure of DNA is shown in Figure 1.1. Viruses are a small exception — the similar molecule RNA instead of DNA is often the genetic material of a virus. In all non-virus organisms, which are composed of cells, each cell contains a full copy of that organism's DNA, called its genome. In the cell, DNA exists as a double-stranded molecule, coiled into the shape of a double-helix. Each nucleotide in DNA preferentially pairs with its partner nucleotide on the opposite strand: A pairs with T, and C pairs with G. Thus, in its two-stranded form, each strand effectively contains all necessary information, redundant with its partner strand. This structure of DNA is the physical basis for inheritance: DNA replication duplicates the genetic information by splitting the strands and using each strand as a template for a partner strand.

Genes express their functional effect through the production of proteins, which are complex molecules responsible for most functions in the cell. Proteins are chains of amino acids, and the DNA sequence of a gene (through an RNA intermediate) is used to produce a specific protein sequence. Each group of three nucleotides in the sequence, also called a codon, corresponds to one of the twenty possibly amino acids

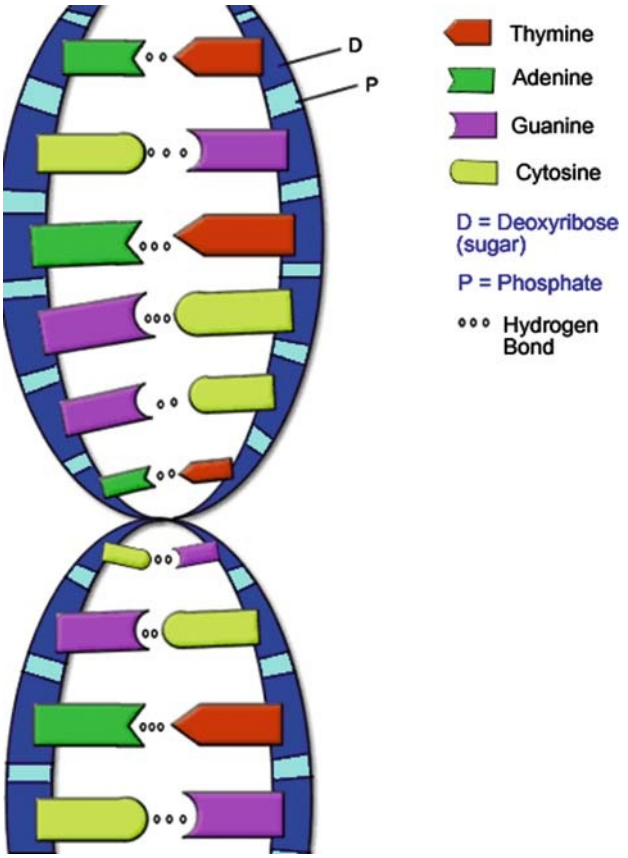


Fig. 1.1 The structure of DNA. Bases pair through the arrangement of hydrogen bonding between the strands

in protein — this correspondence is called the genetic code. The specific sequence of amino acids results in a unique three-dimensional structure for that protein, thereby determining its behavior and function.

1.6 Evolutionary Computation

In computer science evolutionary computation is a subfield of artificial intelligence (more particularly computational intelligence) involving combinatorial optimization problems. It may be recognised by the following criteria:

- iterative progress, growth or development
- population based
- guided random search

- parallel processing
- often biologically inspired

Evolutionary Computation (EC) uses computational models of evolutionary processes as key elements in the design and implementation of computer-based problem solving systems. There are a variety of evolutionary computational models that have been proposed and studied which are referred to as evolutionary algorithms. They share a common conceptual base of simulating the evolution of individual structures via processes of selection and reproduction. These processes depend on the perceived performance (fitness) of the individual structures as defined by an environment. More precisely, evolutionary algorithms maintain a population of structures that evolve according to rules of selection and other operators, such as recombination and mutation. Each individual in the population receives a measure of its fitness in the environment. Selection focuses attention on high fitness individuals, thus exploiting the available fitness information. Recombination and mutation perturb those individuals, providing general heuristics for exploration. Although simplistic from a biologist's viewpoint, these algorithms are sufficiently complex to provide robust and powerful adaptive search mechanisms. Figure 1.2 outlines a typical evolutionary algorithm (EA). A population of individual structures is initialized and then evolved from generation to generation by repeated applications of evaluation, selection, recombination, and mutation. The population size N is generally constant in an evolutionary algorithm, although there is no *a priori* reason (other than convenience) to make this assumption. An evolutionary algorithm typically initializes its population randomly, although domain specific knowledge can also be used to bias the search. Evaluation measures the fitness of each individual according to its worth in some environment. Evaluation may be as simple as computing a fitness function or as complex as running an elaborate simulation. Selection is often performed in two steps, parent selection and survival. Parent selection decides who become parents and how many children the parents have. Children are created via recombination, which exchanges information between parents, and mutation, which further perturbs the children. The children are then evaluated. Finally, the survival step decides who survives in the population.

```
procedure EA; {
t=0;
initialize population P(t);
evaluate P(t);
until (done) {
t=t+1;
parent_selection P(t);
recombine P(t);
mutate P(t);
evaluate P(t);
survive P(t);
} }
```

Fig. 1.2 A typical evolutionary algorithm

1.7 Important Paradigms in Evolutionary Computation

The origins of evolutionary algorithms can be traced to at least the 1950's (e.g., Fraser, 1957; Box, 1957). Three methodologies that have emerged in the last few decades such as: "evolutionary programming (EP)" (L.J. Fogel, A.J. Owens, M.J. Walsh Fogel, 1966), "evolution strategies (ES)" (I. Rechenberg and H.P. Schwefel Rechenberg, 1973), and "genetic algorithms (GA)" (Holland, 1975) are discussed in this section. Although similar at the highest level, each of these varieties implements an evolutionary algorithm in a different manner. The differences touch upon almost all aspects of evolutionary algorithms, including the choices of representation for the individual structures, types of selection mechanism used, forms of genetic operators, and measures of performance. The important differences (and similarities) are also illustrated in the following sections, by examining some of the variety represented by the current family of evolutionary algorithms. These approaches in turn have inspired the development of additional evolutionary algorithms such as "classifier systems (CS)" (Holland, 1986), the LS systems (Smith, 1983), "adaptive operator" systems (Davis, 1989), GENITOR (Whitley, 1989), SAMUEL (Grefenstette, 1989), "genetic programming (GP)" (de Garis, 1990; Koza, 1991), "messy GAs" (Goldberg, 1991), and the CHC approach (Eshelman, 1991). This section will focus on the paradigms such as GA, GP, ES and EP shown in Table 1.2.

1.7.1 Genetic Algorithms

The Genetic Algorithm (GA) is a model of machine learning which derives its behavior from a metaphor of some of the mechanisms of evolution in nature. This is done by the creation within a machine of a population of individuals represented by chromosomes, in essence a set of character strings that are analogous to the base-4 chromosomes. The individuals in the population then go through a process of simulated "evolution".

Genetic algorithms are used for a number of different application areas. An example of this would be multidimensional optimization problems in which the character string of the chromosome can be used to encode the values for the different parameters being optimized.

Table 1.2 Paradigms in evolutionary computation

Paradigm	Created by
Genetic Algorithms	J.H. Holland
Genetic Programming	de Garis and John Koza
Evolutionary Programming	L.J. Fogel, A.J. Owens, M.J. Walsh
Evolution strategies	I. Rechenberg and H.P. Schwefel

In practice, the genetic model of computation can be implemented by having arrays of bits or characters to represent the chromosomes. Simple bit manipulation operations allow the implementation of crossover, mutation and other operations. Although a substantial amount of research has been performed on variable- length strings and other structures, the majority of work with genetic algorithms is focussed on fixed-length character strings. The users should focus on both this aspect of fixed-lengthness and the need to encode the representation of the solution being sought as a character string, since these are crucial aspects that distinguish genetic programming, which does not have a fixed length representation and there is typically no encoding of the problem.

When the genetic algorithm is implemented it is usually done in a manner that involves the following cycle: Evaluate the fitness of all of the individuals in the population. Create a new population by performing operations such as crossover, fitness-proportionate reproduction and mutation on the individuals whose fitness has just been measured. Discard the old population and iterate using the new population.

One iteration of this loop is referred to as a generation. There is no theoretical reason for this as an implementation model. Indeed, behavior in populations in nature is not found as a whole, but it is a convenient implementation model.

The first generation (generation 0) of this process operates on a population of randomly generated individuals. From there on, the genetic operations, in concert with the fitness measure, operate to improve the population.

Algorithm of GA is

```
//start with an initial time
  t :=0;

//initialize a usually random population
  of individuals
  initpopulation P (t);

//evaluate fitness of all initial individuals
  of population
  evaluate P (t);

//test for termination criterion (time, fitness, etc.)
  while not done do

   //increase the time counter
     t :=t+1;

   //select a sub-population for offspring production
     P' :=selectparents P (t);

   //recombine the ''genes'' of selected parents
     recombine P' (t);

   //perturb the mated population stochastically
     mutate P' (t);

   //evaluate its new fitness
     evaluate P' (t);
```

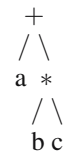
```
    //select the survivors from actual fitness
      P :=survive P,P' (t);
  od
end GA.
```

1.7.2 Genetic Programming

Genetic programming was developed to allow automatic programming and program induction. It may be viewed as a specialized form of genetic algorithm, which manipulates with variable length chromosomes (i.e. with a specialized representation) using modified genetic operators. Unlike genetic algorithms, genetic programming does not distinguish between the search space and the representation space. However, it is not difficult to introduce genotype/phenotype mapping for any evolutionary algorithm formally (it could be one-to-one mapping in its simplest form).

The genetic programming search space includes not only the problem space but also the space of representation of the problem, i.e. genetic programming is able to evolve representations. The theoretical search space of genetic programming is the search space of all possible (recursive) compositions over a primitive set of symbols which the programs are constructed over. The programs are represented either as trees or in a linear form (e.g. machine-language instructions). Crossover is considered as a major operator for genetic programming. It interchanges randomly chosen subtrees of the parents' trees without the syntax of the programs being disrupted. Mutation picks a random subtree and replaces it by a randomly generated one. Genetic programming traditionally evolves symbolic expressions in a functional language like LISP. However, any useful structure may be utilized nowadays. An evolved program can contain code segments which when removed from the program would not alter the result produced by the program (e.g. the instruction $a = a + 0$), i.e. semantically redundant code segments. Such segments are referred to as introns. The size of the evolved program can also grow uncontrollably until it reaches the maximum tree depth allowed while the fitness remains unchanged. This effect is known as bloat. The bloat is a serious problem in genetic programming, since it usually leads to time consuming fitness evaluation and reduction of the effect of search operators. Once it occurs, the fitness almost always stagnates.

These programs are expressed in genetic programming as parse trees, rather than as lines of code. Thus, for example, the simple program "$a + b^*c$" would be represented as follows



or, to be precise, as suitable data structures linked together to achieve this effect. Because this is a very simple thing to do in the programming language Lisp, many GP users tend to use Lisp. However, this is simply an implementation detail. There are straightforward methods to implement GP using a non-Lisp programming environment.

The programs in the population are composed of elements from the function set and the terminal set, which are typically fixed sets of symbols selected to be appropriate to the solution of problems in the domain of interest. In GP the crossover operation is implemented by taking randomly selected subtrees in the individuals (selected according to fitness) and exchanging them. It should be pointed out that GP usually does not use any mutation as a genetic operator.

Genetic programming can also employ a number of advanced genetic operators. For instance, automatically defined functions (ADF) allow the definition of subprograms that can be called from the rest of the program. Then the evolution is to find the solution as well as its decomposition into ADFs together. The fitness function is either application specific for a given environment or it takes the form of symbolic regression. In all cases, the evolved program must be executed in order to find out what it does. The outputs of the program are usually compared with the desired outputs for given inputs. A terminating mechanism of the fitness evaluation process must be introduced to stop the algorithm. Figure 1.3 outlines a typical genetic programming (GP) algorithm.

1.7.3 Evolutionary Programming

Evolutionary Programming (EP), originally conceived by Lawrence J. Fogel in 1960, is a stochastic optimization strategy similar to genetic algorithms, but instead places emphasis on the behavioral linkage between parents and their offspring, rather than seeking to emulate specific genetic operators as observed in nature. Evolutionary programming is similar to evolution strategies, although the two approaches developed independently. Like both ES and GAs, EP is a useful method

```
procedure GP;
{
t=0;
initialize population P(t);
evaluate P(t);
until (done) {
t=t+1;
parent_selection P(t);
crossover P(t)
mutate P(t);
evaluate P(t);
survive P(t);
} }
```

Fig. 1.3 The GP algorithm

of optimization when other techniques such as gradient descent or direct, analytical discovery are not possible. Combinatory and real-valued function optimization in which the optimization surface or fitness landscape is "rugged", possessing many locally optimal solutions, are well suited for evolutionary programming.

For EP, like GAs, there is an underlying assumption that a fitness landscape can be characterized in terms of variables, and that there is an optimum solution (or multiple such optima) in terms of those variables. For example, if one were trying to find the shortest path in a Traveling Salesman Problem, each solution would be a path. The length of the path could be expressed as a number, which would serve as the solution's fitness. The fitness landscape for this problem could be characterized as a hyper surface proportional to the path lengths in a space of possible paths. The goal would be to find the globally shortest path in that space, or more practically, to find very short tours very quickly.

The basic EP method involves 3 steps (Repeat until a threshold for iteration is exceeded or an adequate solution is obtained):

(1) Choose an initial population of trial solutions at random. The number of solutions in a population is highly relevant to the speed of optimization, but no definite answers are available as to how many solutions are appropriate (other than > 1) and how many solutions are just wasteful.
(2) Each solution is replicated into a new population. Each of these offspring solutions are mutated according to a distribution of mutation types, ranging from minor to extreme with a continuum of mutation types between. The severity of mutation is judged on the basis of the functional change imposed on the parents.
(3) Each offspring solution is assessed by computing its fitness. Typically, a stochastic tournament is held to determine N solutions to be retained for the population of solutions, although this is occasionally performed deterministically. There is no requirement that the population size be held constant, nor that only a single offspring be generated from each parent. It should be pointed out that EP typically does not use any crossover as a genetic operator.

Algorithm of EP is

```
//start with an initial time
  t :=0;

//initialize a usually random population
  of individuals
  initpopulation P(t);

//evaluate fitness of all initial individuals
  of population
  evaluate P(t);

//test for termination criterion (time, fitness, etc.)
  while not done do

    //perturb the whole population stochastically
      P'(t) :=mutate P (t);
```

```
    //evaluate its new fitness
      evaluate P'(t);

    //stochastically select the survivors from actual
      fitness
      P(t+1) :=survive P(t),P'(t);

    //increase the time counter
      t :=t+1;
  od
end EP
```

1.7.4 Evolution Strategies

Evolution strategies (ESs) were independently developed with selection, mutation, and a population of size one. Schwefel introduced recombination and populations with more than one individual, and provided a nice comparison of ESs with more traditional optimization techniques. Due to initial interest in hydrodynamic optimization problems, evolution strategies typically use real-valued vector representations.

Thus, Evolution Strategies were invented to solve technical optimization problems (TOPs) like e.g. constructing an optimal flashing nozzle, and until recently ES were only known to civil engineering folks, as an alternative to standard solutions. Usually no closed form analytical objective function is available for TOPs and hence, no applicable optimization method exists, but the engineer's intuition.

The first attempts to imitate principles of organic evolution on a computer still resembled the iterative optimization methods known up to that time: In a two-membered ES, one parent generates one offspring per generation by applying normally distributed mutations, i.e. smaller steps occur more likely than big ones, until a child performs better than its ancestor and takes its place. Because of this simple structure, theoretical results for step size control and convergence velocity could be derived. The ratio between successful and all mutations is 1:5, the so-called 1/5 success rule was discovered. This first algorithm, using mutation only, has then been enhanced to a $(m+1)$ strategy which incorporated recombination due to several, i.e. m parents being available. The mutation scheme and the exogenous step size control were taken across unchanged from two-membered ESs.

Schwefel later generalized these strategies to the multi-membered ES now denoted by $(m+l)$ and (m,l) which imitates the following basic principles of organic evolution: a population, leading to the possibility of recombination with random mating, mutation and selection. These strategies are termed plus strategy and comma strategy, respectively: in the plus case, the parental generation is taken into account during selection, while in the comma case only the offspring undergoes selection, and the parents die off. m (usually a lowercase m, denotes the population size, and l, usually a lowercase *lambda* denotes the number of offspring generated per generation). The algorithm of an evolutionary strategy is as follows:

```
(define (Evolution-strategy population)
 (if (terminate? population)
  population
   (evolution-strategy
    (select
     (cond (plus-strategy?
            (union (mutate
                    (recombine population))
                   population))
       (comma-strategy?
        (mutate
         (recombine population)))))))))
```

A single individual of the ES' population consists of the following genotype representing a point in the search space:

Object Variables

Real-valued x(i) have to be tuned by recombination and mutation such that an objective function reaches its global optimum.

Strategy Variables

Real-valued s(i) (usually denoted by a lowercase sigma) or mean stepsizes determine the mutability of the x(i). They represent the standard deviation of a (0, s(i)) gaussian distribution (GD) being added to each x(i) as an undirected mutation. With an "expectancy value" of 0 the parents will produce offspring similar to themselves on average. In order to make a doubling and a halving of a stepsize equally probable, the s(i) mutate log-normally, distributed, i.e. exp(GD), from generation to generation. These stepsizes hide the internal model the population has made of its environment, i.e. a self-adaptation of the stepsizes has replaced the exogenous control of the $(1+1)$ ES.

This concept works because selection sooner or later prefers those individuals having built a good model of the objective function, thus producing better offspring. Hence, learning takes place on two levels: (1) at the genotypic, i.e. the object and strategy variable level and (2) at the phenotypic level, i.e. the fitness level.

Depending on an individual's x(i), the resulting objective function value f(x), where x denotes the vector of objective variables, serves as the phenotype (fitness) in the selection step. In a plus strategy, the m best of all $(m+l)$ individuals survive to become the parents of the next generation. Using the comma variant, selection takes place only among the l offspring. The second scheme is more realistic and therefore more successful, because no individual may survive forever, which could at least theoretically occur using the plus variant. Untypical for conventional optimization algorithms and lavish at first sight, a comma strategy allowing intermediate

deterioration performs better! Only by *forgetting* highly fit individuals can a permanent adaptation of the stepsizes take place and avoid long stagnation phases due to misadapted sizes. This means that these individuals have built an internal model that is no longer appropriate for further progress, and thus should better be discarded.

By choosing a certain ratio m/l, one can determine the convergence property of the evolution strategy: If one wants a fast, but local convergence, one should choose a small hard selection, ratio, e.g. (5,100), but looking for the global optimum, one should favour a softer selection (15,100).

Self-adaptation within ESs depends on the following agents:

Randomness

One cannot model mutation as a purely random process. This would mean that a child is completely independent of its parents.

Population Size

The population has to be sufficiently large. Not only should the current best be allowed to reproduce, but a set of good individuals. Biologists have coined the term "requisite variety" to mean the genetic variety necessary to prevent a species from becoming poorer and poorer genetically and eventually dying out.

Cooperation

In order to exploit the effects of a population $(m > 1)$, the individuals should recombine their knowledge with that of others (cooperate) because one cannot expect the knowledge to accumulate in the best individual only.

Deterioration

In order to allow better internal models (step sizes) to provide better progress in the future, one should accept deterioration from one generation to the next. A limited life-span in nature is not a sign of failure, but an important means of preventing a species from freezing genetically.

ESs proves to be successful when compared to other iterative methods on a large number of test problems. They are adaptable to nearly all sorts of problems in optimization, because they need very little information about the problem, especially no derivatives of the objective function. ESs are capable of solving *high dimensional, multimodal, nonlinear problems* subject to *linear and/or nonlinear constraints*. The objective function can also, e.g. be the result of a simulation, it does not have to be given in a closed form. This also holds for the constraints which may represent the outcome of, e.g. a finite elements method (FEM). ESs have been adapted to vector optimization problems, and they can also serve as a heuristic for NP-complete combinatorial problems like the travelling salesman problem or problems with a noisy or changing response surface.

1.8 Global Optimization

Global optimization is the task of finding the absolutely best set of parameters to optimize an objective function. In general, there can solutions that are locally optimal but not globally optimal. Consequently, global optimization problems are typically quite difficult to solve exactly; in the context of combinatorial problems, they are often NP-hard. Global optimization problems fall within the broader class of nonlinear programming (NLP). Methods for global optimization problems can be categorized based on the properties of the problem that are used and the types of guarantees that the methods provide for the final solution. A NLP has the form

$$\begin{aligned} &\text{minimize } F(x) \\ &\text{subject to } g_i(x) = 0 \qquad \text{for } i = 1,\ldots,m1 \qquad \text{where } m1 >= 0 \\ &\qquad\qquad h_j(x) >= 0 \quad \text{for } j = m1+1,\ldots,m \quad \text{where } m >= m1 \end{aligned}$$

where F is a function of a vector of reals x that is subject to equality and inequality constraints. Some of the most important classes of global optimization problems are differential convex optimization, complementary problems, min-max problems, bi-linear and bi-convex programming, continuous global optimization and quadratic programming.

Specific optimization methods have been developed for many classes of global optimization problems. Additionally, general techniques have been developed that appear to have applicability to a wide range of problems.

- **Combinatorial problems** have a linear or nonlinear function defined over a set of solutions that is finite but very large. There are a number of significant categories of combinatorial optimization problems, including network problems, scheduling, and transportation. If the function is piecewise linear, the combinatorial problem can be solved exactly with a mixed integer program method, which uses branch and bound. Heuristic methods like simulated annealing, Tabu search and genetic algorithms have also been successfully applied to these problems to find approximate solutions.
- **General unconstrained problems** have a nonlinear function over reals that is unconstrained (or which have simple bound constraints). A variety of partitioning strategies have been proposed to solve this problem exactly. These methods typically rely on *a priori* knowledge of how rapidly the function can vary (e.g. the Lipshitz constant) or the availability of an analytic formulation of the objective function (e.g. interval methods). Statistical methods also use partitioning to decompose the search space, but they use *a priori* information (or assumptions) about how the objective function can be modeled. A wide variety of other methods have been proposed for solving these problems inexactly, including simulated annealing, genetic algorithms, clustering methods, and continuation methods, which first transform the potential function into a smoother function

with fewer local minimizers, and then uses a local minimization procedure to trace the minimizers back to the original function.

- **General constrained problems** have a nonlinear function over reals that are constrained. These types of problems have not received as much attention as have general unconstrained problems. However, many of the methods for unconstrained problems have been adapted to handle constraints.

1.9 Techniques of Global Optimization

Some of the global optimization techniques are discussed in this section as follows:

1.9.1 Branch and Bound

Branch and Bound is a general search method. Suppose it is required to minimize a function f(x), where x is restricted to some feasible region (defined, e.g., by explicit mathematical constraints). To apply branch and bound, a means of computing a lower bound on an instance of the optimization problem and a means of dividing the feasible region of a problem to create smaller sub problems must be available. There must also be a way to compute an upper bound (feasible solution) for at least some instances; for practical purposes, it should be possible to compute upper bounds for some set of nontrivial feasible regions.

The method starts by considering the original problem with the complete feasible region, which is called the root problem. The lower-bounding and upper-bounding procedures are applied to the root problem. If the bounds match, then an optimal solution has been found and the procedure terminates. Otherwise, the feasible region is divided into two or more regions, each strict sub regions of the original, which together cover the whole feasible region; ideally, these sub problems partition the feasible region. These sub problems become children of the root search node. The algorithm is applied recursively to the sub problems, generating a tree of sub problems. If an optimal solution is found to a sub problem, it is a feasible solution to the full problem, but not necessarily globally optimal. Since it is feasible, it can be used to prune the rest of the tree: if the lower bound for a node exceeds the best known feasible solution, no globally optimal solution can exist in the subspace of the feasible region represented by the node.

Therefore, the node can be removed from consideration. The search proceeds until all nodes have been solved or pruned, or until some specified threshold is meet between the best solution found and the lower bounds on all unsolved sub problems.

1.9.2 Clustering Methods

Clustering global optimization methods can be viewed as a modified form of the standard multi-start procedure, which performs a local search from several points distributed over the entire search domain. A drawback of multi-start is that when many starting points are used the same local minimum may identified several times, thereby leading to an inefficient global search. Clustering methods attempt to avoid this inefficiency by carefully selecting points at which the local search is initiated. The three main steps of clustering methods are: (1) sample points in the search domain, (2) transform the sampled point to group them around the local minima, and (3) apply a clustering technique to identify groups that (hopefully) represent neighborhoods of local minima. If this procedure successfully identifies groups that represent neighborhoods of local minima, then redundant local searches can be avoided by simply starting a local search for some point within each cluster.

Clustering methods have been developed for optimizing unconstrained functions over reals. These methods assume that the objective function is relatively inexpensive since many points are randomly sampled to identify the clusters. Clustering methods are most effective for low dimensional problems, so these methods become less effective for problems of more than a few hundred variables.

1.9.3 Hybrid Methods

There are a wide range of hybrid global optimization algorithms that have been developed. Several hybrid algorithms are:

- MINLP
- Tree Annealing

MINLP (Mixed Integer Nonlinear Programming)

Mixed Integer Nonlinear Programming (MINLP) refers to mathematical programming algorithms that can optimize both continuous and integer variables, in a context of nonlinearities in the objective function and/or constraints. The general form of a MINLP is:

$$
\begin{aligned}
&\mathrm{Z} = \min \mathrm{C}(\mathrm{y},\mathrm{x}) \text{ w.r.t.}(\mathrm{y},\mathrm{x}) \\
&\text{s.t. } \mathrm{h}(\mathrm{y},\mathrm{x}) = 0 \\
&\quad \mathrm{g}(\mathrm{y},\mathrm{x}) < 0 \\
&\mathrm{y} \text{ in} \{0,1\}^{\mathrm{m}}, \mathrm{x} \text{ in } \mathrm{R}^{\mathrm{n}}
\end{aligned}
$$

Engineering design problems often are MINLP problems, since they involve the selection of a configuration or topology (which components should be included in the design) as well as the design parameters of those components, i.e., size, weight, etc. The inclusion of a component can be represented by the binary variables, whereas the design parameters are continuous. Usually, the 0–1 variables appear in a linear form and the continuous variables exhibit the nonlinearities.

MINLP problems are NP-complete and until recently have been considered extremely difficult. However, with current problem structuring methods and computer technology, they are now solvable. Major algorithms for solving the MINLP problem include: branch and bound, generalized Benders decomposition (GBD), and outer approximation (OA).

The branch and bound method of solution is an extension of B&B for mixed integer programming. The method starts out by relaxing the integrality requirements of the 0–1 variables, thus forming a continuous NLP problem. Then, a tree enumeration is performed in which a subset of the integer variables has successively fixed at each node. The solution of the corresponding NLP at each node provides a lower bound for the optimal MINLP objective function value. The lower bound is used to direct the search, either by expanding the nodes in a breadth first or depth first enumeration, depending on the value of the lower bound. The major disadvantage of the branch and bound method is that it may require the solution of a large number of NLP sub problems (depending upon the number of integer variables).

Generalized Benders decomposition (GBD) and outer approximation solve the MINLP by an iterative process. The problem is decomposed into a NLP sub problem (which has the integer values fixed) and an MINLP master problem. The NLP sub-problems optimize the continuous variables and provide an upper bound to the MINLP solution, while the MILP master problems have the role of predicting a new lower bound for the MINLP solution, as well as new integer variables for each iteration. The search terminates when the predicted lower bound equals or exceeds the current upper bound. The process is shown in Figure 1.4.

In terms of algorithm performance and computational experience with MINLP, very large problems (hundreds of 0–1 variables, thousands of continuous variables, and thousands of constraints) have been solved. No single solution method seemed to be consistently superior in all applications.

Application Domains

MINLP problems involve the simultaneous optimization of discrete and continuous variables. These problems often arise in engineering domains, where one is trying to simultaneously optimize the system structure and parameters. This is difficult because optimal topology is dependent upon parameter levels and vice versa. In many design optimization problems, the structural topology influences the optimal parameter settings so a simple de-coupling approach does not work: it is often not possible to isolate these and optimize each separately. Finally, the complexity of these problems depends upon the form of the objective function. In the past, solution

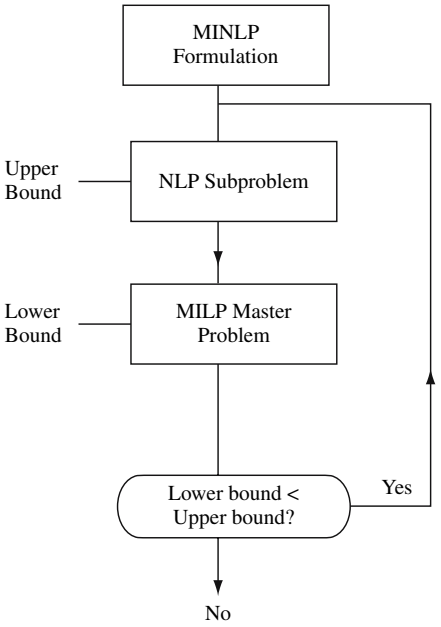


Fig. 1.4 Decomposition method

techniques typically depended upon objective functions that were single-attribute and linear (i.e., minimize cost). However, real problems often require multi-attribute objectives such as minimizing costs while maximizing safety and/or reliability, ensuring feasibility, and meeting scheduled deadlines. In these cases, the goal is to optimize over a set of performance indices which may be combined in a nonlinear objective function.

Engineering design "synthesis" problems are a major application of MINLP algorithms. In engineering design, one often wants to know how to select a topology or configuration and the parameters of the system to meet the specified goals and functional requirements. Synthesis is defined as "In-order to select the configuration or topology, as well as its design parameters, one has to determine which components should integrate a system and how they should be connected; on the other hand, one has to determine the sizes and parameters of the components. The former clearly implies making discrete decisions, while the latter implies making a choice from among a continuous space."

Examples of synthesis problems include the design of structures in civil engineering, of process flow sheets in chemical engineering, of VLSI circuits in electrical engineering, of servomechanisms in mechanical engineering, etc. To formulate a synthesis problem, a superstructure is postulated that has all the possible alternatives that are candidates for a feasible design embedded in it. The discrete variables are the decision variables for the components in the superstructure to include in the

optimal structure, and the continuous variables are the values of the parameters for the included components.

Tree Annealing

Simulated annealing (pointer to web page) was designed for combinatorial optimization, usually implying that the decision variables are discrete. A variant of simulated annealing called tree annealing was developed by Bilbro and Snyder to globally minimize continuous functions.

In simulated annealing, one is given an initial point x and randomly generates a new candidate solution, y. If $f(y) < f(x)$, then y replaces x and the process repeats. If $f(y) > f(x)$, y replaces x with a probability that is based on the Metropolis criteria and the progress of the algorithm. In tree annealing, candidate subspaces are accepted or rejected by evaluating the function at representative points. Tree annealing gradually narrows the width of the subspace to close in around the global minimum. However, tree annealing does not find the global minimizer x^* exactly: it finds an arbitrarily small interval where that minimum lies. The tree annealing algorithm is very similar to simulated annealing: given a point x associated with the current search interval and a point y associated with a candidate search interval, y replaces x depending upon $f(x)$, $f(y)$, and the progress of the algorithm.

Tree annealing stores information in a binary tree to keep track of which subintervals have been explored. Each node in the tree represents one of two subintervals defined by the parent node. Initially the tree consists of one parent and two child nodes. As better subintervals are found, the path down the tree that leads to these intervals gets deeper and the nodes along this path define smaller and smaller subspaces. The tree annealing algorithm is as follows:

1. Randomly choose an initial point x over the search interval S_o
2. Randomly travel down the tree to an arbitrary terminal node i, and generate a candidated point y over the subspace defined by S_i.
3. If $f(y) < F(X)$ replace x with y, and go to step 5.
4. Compute $P = \exp(-(f(y) - f(x))/T)$. If $P > R$, where R is a random number uniformly distributed between 0 and 1, replace x with y.
5. If y replace x, decrease T slightly and update the tree until $T < T_{min}$.

Application Domains

Tree annealing is a method for finding the global minimum of non-convex functions. It is an extension of simulated annealing in that it can handle continuous variables. It is not guaranteed to converge, and often the rate of convergence is very slow. It is recommended to use tree annealing in the first part of optimization to find the region of the global optimum, then use a gradient descent or other method to hone in on the actual value. Given enough time, a sufficiently low final temperature, and a slow cooling rate, tree annealing should at least converge to a local minimum, and

will often converge to a global minimum, especially for piecewise functions with discontinuities.

Tree annealing has been used in applications such as signal processing and spectral analysis, neural nets, data fitting, etc. These problems involve fitting parameters to noisy data, and often it is difficult to find an optimal set of parameters via conventional means.

1.9.4 Simulated Annealing

Simulated annealing is a generalization of a Monte Carlo method for examining the equations of state and frozen states of n-body systems. The concept is based on the manner in which liquids freeze or metals re-crystallize in the process of annealing. In an annealing process a melt, initially at high temperature and disordered, is slowly cooled so that the system at any time is approximately in thermodynamic equilibrium. As cooling proceeds, the system becomes more ordered and approaches a "frozen" ground state at $T = 0$. Hence the process can be thought of as an adiabatic approach to the lowest energy state. If the initial temperature of the system is too low or cooling is done insufficiently slowly the system may become quenched forming defects or freezing out in meta-stable states (ie.,. trapped in a local minimum energy state).

The original Metropolis scheme was that an initial state of a thermodynamic system was chosen at energy E and temperature T, holding T constant the initial configuration is perturbed and the change in energy dE is computed. If the change in energy is negative the new configuration is accepted. If the change in energy is positive it is accepted with a probability given by the Boltzmann factor $exp - (dE/T)$. This processes is then repeated sufficient times to give good sampling statistics for the current temperature, and then the temperature is decremented and the entire process repeated until a frozen state is achieved at $T = 0$.

By analogy the generalization of this Monte Carlo approach to combinatorial problems is straight forward. The current state of the thermodynamic system is analogous to the current solution to the combinatorial problem, the energy equation for the thermodynamic system is analogous to at the objective function, and ground state is analogous to the global minimum. The major difficulty (art) in implementation of the algorithm is that there is no obvious analogy for the temperature T with respect to a free parameter in the combinatorial problem. Furthermore, avoidance of entrainment in local minima (quenching) is dependent on the "annealing schedule", the choice of initial temperature, how many iterations are performed at each temperature, and how much the temperature is decremented at each step as cooling proceeds.

Application Domains

Simulated annealing has been used in various combinatorial optimization problems and has been particularly successful in circuit design problems.

1.9.5 Statistical Global Optimization Algorithms

Statistical global optimization algorithms employ a statistical model of the objective function to bias the selection of new sample points. These methods are justified with Bayesian arguments that suppose that the particular objective function that is being optimized comes from a class of functions that is modeled by a particular stochastic function. Information from previous samples of the objective function can be used to estimate parameters of the stochastic function, and this refined model can subsequently be used to bias the selection of points in the search domain.

This framework is designed to cover average conditions of optimization. One of the challenges of using statistical methods is the verification that the statistical model is appropriate for the class of problems to which they are applied. Additionally, it has proved difficult to devise computationally interesting versions of these algorithms for high dimensional optimization problems.

Application Domains

Virtually all statistical methods have been developed for objective functions defined over the reals. Statistical methods generally assume that the objective function is sufficiently expensive that it is reasonable for the optimization method to perform some nontrivial analysis of the points that have been previously sampled. Many statistical methods rely on dividing the search region into partitions. In practice, this limits these methods to problems with a moderate number of dimensions.

Statistical global optimization algorithms have been applied to some challenging problems. However, their application has been limited due to the complexity of the mathematical software needed to implement them.

1.9.6 Tabu Search

The basic concept of Tabu Search as described by Glover (1986) is “a meta-heuristic superimposed on another heuristic”. The overall approach is to avoid entrainment in cycles by forbidding or penalizing moves which take the solution, in the next iteration, to points in the solution space previously visited (hence “Tabu”). The Tabu search is fairly new, Glover attributes it’s origin to about 1977. The method is still actively researched, and is continuing to evolve and improve. The Tabu method was

partly motivated by the observation that human behavior appears to operate with a random element that leads to inconsistent behavior given similar circumstances. As Glover points out, the resulting tendency to deviate from a charted course, might be regretted as a source of error but can also prove to be source of gain. The Tabu method operates in this way with the exception that new courses are not chosen randomly. Instead the Tabu search proceeds according to the supposition that there is no point in accepting a new (poor) solution unless it is to avoid a path already investigated. This insures new regions of a problems solution space will be investigated in with the goal of avoiding local minima and ultimately finding the desired solution.

The Tabu search begins by marching to a local minimum. To avoid retracing the steps used the method records recent moves in one or more Tabu lists. The original intent of the list was not to prevent a previous move from being repeated, but rather to insure it was not reversed. The Tabu lists are historical in nature and form the Tabu search memory. The role of the memory can change as the algorithm proceeds. At initialization the goal is make a coarse examination of the solution space, known as 'diversification', but as candidate locations are identified the search is more focused to produce local optimal solutions in a process of 'intensification'. In many cases the differences between the various implementations of the Tabu method have to do with the size, variability, and adaptability of the Tabu memory to a particular problem domain.

Application Domain

The Tabu search has traditionally been used on combinatorial optimization problems. The technique is straightforwardly applied to continuous functions by choosing a discrete encoding of the problem. Many of the applications in the literature involve integer programming problems, scheduling, routing, traveling salesman and related problems.

1.9.7 Multi Objective Optimization

When tackling real-world problems, particularly in the field of engineering design, the desired optimal design may not he expressed in terms of a single objective. Product designers may wish to maximize some element-of the performance of a product, while minimizing the cost of its manufacture, for example. Different objectives may be conflicting, with little a priori knowledge as to how they interact. In the past, common practice was to optimize for a single objective while applying other objectives as penalty functions or constraints, a less than ideal approach. Evolutionary Algorithms (EA) have recently become more widely used for their ability to work well with large-scale, multi-objective problems.

In general, there remains a role for a human Decision Maker (DM) in choosing between competing objectives. Multi-objective optimization algorithms can broadly be categorized by when in the optimization process the DM intervenes:

Decision Making Before Search

The DM aggregates the objectives into a single objective, including preference information (or weights). The problem is essentially reduced to a single objective optimization.

Decision Making During Search

The DM interactively supplies preference information to guide the search.

Decision Making After Search

The DM selects from a set of candidate solutions resulting from the search.

It may be noted that the first two of these approaches would seem to require, some a priori knowledge of the problem domain in order to effectively provide preference information, particularly for methods involving aggregation. It has been asserted. that the first approach does not return Pareto-optimal solutions in the presence of non-convex search spaces. EAs applied to multi-objective optimization need to address two main problems:

- How is fitness assignment and selection performed?
- How is a diverse population maintained, to aid search space exploration?

Fitness assignment can conceptually he divided into a number of approaches such as:

- Aggregation-based
- Criterion-based
- Pareto-based

Evolutionary Algorithms, however, have been recognized to be possibly well-suited to multi-objective optimization since early in their development. Multiple individuals can search for multiple solutions in parallel, eventually taking advantage of any similarities available in the family of possible solutions to the problem. The ability to handle complex problems involving features such as discontinuities, multimodality, disjoint feasible spaces and noisy function evaluations, reinforces the potential effectiveness of EAs in multi-objective search and optimization, which is perhaps a problem area where EC really distinguishes itself from its competitors.

Summary

Evolutionary algorithm is an umbrella term used to describe computer based problem solving systems which use computational models of some of the known mechanisms of evolution as key elements in their design and implementation. Introduction to the different types of evolutionary algorithms were discussed in this chapter. Methods for global optimization problems were also categorized based on the properties of the problem that are used and the types of guarantees that the methods provide for the final solution. Major application domains of the optimization were also discussed in this chapter.

Review Questions

1. Write a short note on Natural Evolution.
2. Compare biological terms with the EC terms.
3. Explain Darwin's "descent with modification".
4. What is meant by natural selection?
5. Mention the inferences of Darwin's theory of evolution.
6. Explain Darwin's theory of evolution in brief.
7. Define "Genetics".
8. What is Evolutionary Computation?
9. Describe the basic EC algorithm.
10. Mention the paradigms of Evolutionary Computation.
11. What is Genetic Algorithm? Explain its algorithm.
12. What is Genetic Programming? Explain its algorithm.
13. What is Evolutionary Programming? Explain its algorithm.
14. What are Evolutionary Strategies? Explain its algorithm.
15. What are the basic steps of operation in Evolutionary Programming?
16. What is the need for global optimization?
17. Mention a few areas of application where global optimization is required?
18. Mention the different techniques of global optimization?
19. What are the steps performed in clustering method?
20. Differentiate MINLP and Tree annealing?
21. Mention a few application domains of simulated annealing?
22. Explain the basic concept of Tabu search.
23. Mention a few application domains of Tabu search.
24. What are the different approaches in which fitness is assigned to multi-objective optimization?

Chapter 2
Principles of Evolutionary Algorithms

Learning Objectives: On completion of this chapter the reader will have knowledge on:

- Structure of a single population evolutionary algorithm
- Components of Evolutionary algorithms
- Various representation techniques
  - Fitness Function
  - Population Initialization
  - Selection mechanisms
  - Recombination methods
  - Mutation techniques
  - Reinsertion methods
  - Reproduction operator
- Categorization of Parallel Evolutionary Algorithms
- Advantages of evolutionary algorithms
- Multi-objective evolutionary algorithms

2.1 Introduction

Evolutionary algorithms (EAs) are modern techniques for searching complex spaces for an optimum. These meta-heuristics can be differentiated into a quite large set of algorithmic families representing bio-inspired systems, which mimic natural evolution. They are said to be "evolutionary" because their basic behavior is drawn from nature, and evolution is the basic natural force driving a population of tentative solutions towards the problem regions where the optimal solutions are located.

Evolutionary algorithms operate on a population of potential solutions applying the principle of survival of the fittest to produce better and better approximations to a solution. At each generation, a new set of approximations is created by the process

of selecting individuals according to their level of fitness in the problem domain and breeding them together using operators borrowed from natural genetics. This process leads to the evolution of populations of individuals that are better suited to their environment than the individuals that they were created from, just as in natural adaptation.

Evolutionary algorithms (EAs) have become the method of choice for optimization problems that are too complex to be solved using deterministic techniques such as linear programming or gradient Jacobian methods. The large number of applications and the continuously growing interest in this field are due to several advantages of EAs compared to gradient-based methods for complex problems. This chapter provides an insight into evolutionary algorithms and its components. EAs often take less time to find the optimal solution than gradient methods. However, most real-world problems involve simultaneous optimization of several often mutually concurrent objectives. Multi-objective EAs are able to find optimal trade-offs in order to get a set of solutions that are optimal in an overall sense. This chapter also discusses multi-objective optimization using evolutionary algorithms.

2.2 Structure of Evolutionary Algorithms

Evolutionary algorithms model natural processes, such as selection, recombination, mutation, migration, locality and neighborhood. Figure 2.1 shows the structure of a simple evolutionary algorithm. Evolutionary algorithms work on populations of individuals instead of single solutions. In this way the search is performed in a parallel manner.

At the beginning of the computation a number of individuals (the population) are randomly initialized. The objective function is then evaluated for these individuals. The first/initial generation is produced.

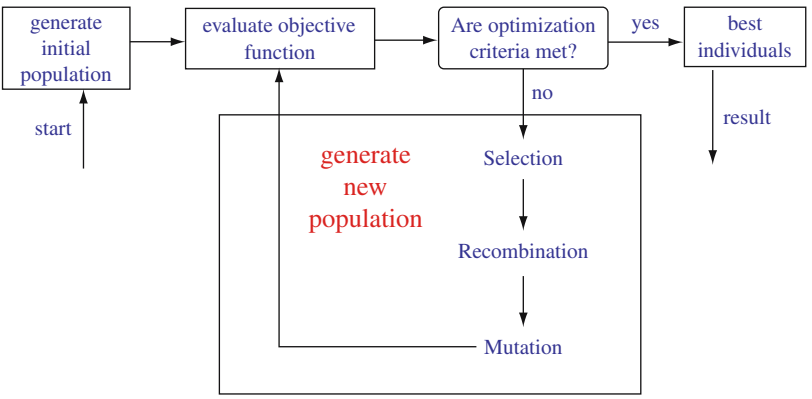


Fig. 2.1 Structure of a single population evolutionary algorithm

If the optimization criteria are not met with the creation of a new generation starts. Individuals are selected according to their fitness for the production of offspring. Parents are recombined to produce offspring. All offsprings will be mutated with a certain probability. The fitness of the offspring is then computed. The offsprings are inserted into the population replacing the parents, producing a new generation. This cycle is performed until the optimization criteria are reached. Such a single population evolutionary algorithm is powerful and performs well on a wide variety of problems. However, better results can be obtained by introducing multiple subpopulations. Every subpopulation evolves over a few generations isolated (like the single population evolutionary algorithm) before one or more individuals are exchanged between the subpopulation. The multi-population evolutionary algorithm models the evolution of a species in a way more similar to nature than the single population evolutionary algorithm. Figure 2.2 shows the structure of such an extended multi-population evolutionary algorithm.

From the above discussion, it can be seen that evolutionary algorithms differ substantially from more traditional search and optimization methods. The most significant differences are:

- Evolutionary algorithms search a population of points in parallel, not just a single point.
- Evolutionary algorithms do not require derivative information or other auxiliary knowledge; only the objective function and corresponding fitness levels influence the directions of search.
- Evolutionary algorithms use probabilistic transition rules, not deterministic ones.

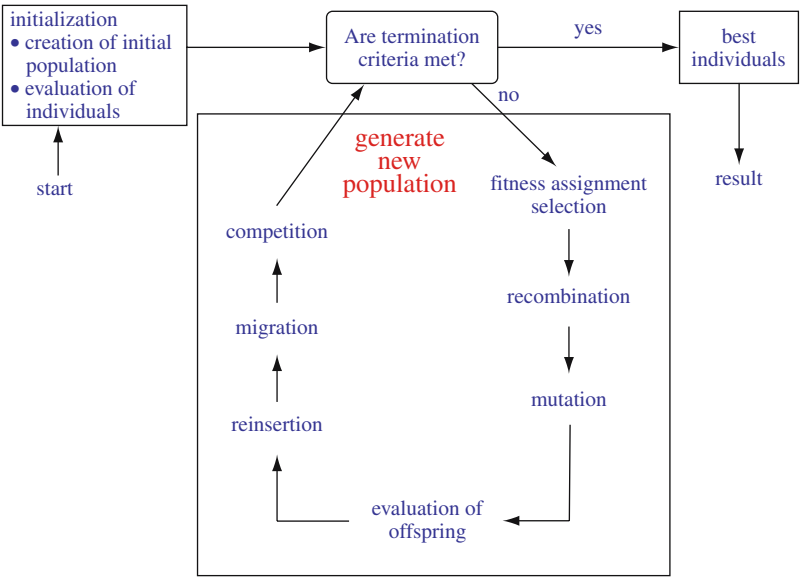


Fig. 2.2 Structure of an extended multi-population evolutionary algorithm

- Evolutionary algorithms are generally more straightforward to apply, because no restrictions for the definition of the objective function exist.
- Evolutionary algorithms can provide a number of potential solutions to a given problem. The final choice is left to the user. (Thus, in cases where the particular problem does not have one individual solution, for example a family of pareto-optimal solutions, as in the case of multi-objective optimization and scheduling problems, then the evolutionary algorithm is potentially useful for identifying these alternative solutions simultaneously.)

2.2.1 Illustration

The evolutionary algorithm is illustrated here with a simple example. Suppose an automotive manufacturer wishes to design a new engine and fuel system in order to maximize performance, reliability, and gas-mileage, while minimizing emissions. It is further assumed that an engine simulation unit can test various engines and return a single value indicating the fitness score of the engine. However, the number of possible engines is large and there is insufficient time to test them all. How would one attack such a problem with an evolutionary algorithm?

First, it is defined that each individual represent a specific engine. For example, suppose the cubic inch displacement (CID), fuel system, number of valves, cylinders, and presence of turbo-charging are all engine variables. The initialization step would create an initial population of possible engines. For the sake of simplicity, assume a (very small) population of size four. Table 2.1 shows an example of initial population:

Now each individual is evaluated with the engine simulator. Each individual receives a fitness score (the higher the better).

Parent selection decides who has children and how many to have. For example, it can be decided that individual 3 deserves two children, because it is much better than the other individuals. Children are created through recombination and mutation. Recombination exchanges information between individuals, while mutation perturbs individuals, thereby increasing diversity. For example, recombination of individuals 3 and 4 could produce the two children 3' and 4' as shown in Table 2.2.

Note that the children are composed of elements of the two parents. Further note that the number of cylinders must be four, because individuals 3 and 4 both had four cylinders. Mutation might further perturb these children, yielding the Table 2.3.

Table 2.1 Initial population

Individual	CID	Fuel System	Turbo	Valves	Cylinders
1	350	4 Barrels	Yes	16	8
2	250	Mech. Inject.	No	12	6
3	150	Elect. Inject.	Yes	12	4
4	200	2 Barrels	No	8	4

Table 2.2 Children

Individual	CID	Fuel System	Turbo	Valves	Cylinders
$3'$	200	Elect. Inject.	Yes	8	4
$4'$	150	2 Barrels	No	12	4

Table 2.3 After mutation

Individual	CID	Fuel System	Turbo	Valves	Cylinders
$3'$	250	Elect. Inject.	Yes	8	4
$4'$	150	2 Barrels	No	12	6

Now the children are evaluated as shown in Table 2.4.

Finally it is decided as to who will survive. In the constant population size example, which is typical of most EAs, four individuals have to be selected for survival. The way in which this is accomplished varies considerably in different EAs. If, for example, only the best individuals survive, the population would become as shown in Table 2.5.

This cycle of evaluation, selection, recombination, mutation, and survival continues until some termination criterion is met.

This simple example serves to illustrate the flavor of an evolutionary algorithm. It is important to point out that although the basic conceptual framework of all EAs is similar, their particular implementations differ in many details. For example, there are a wide variety of selection mechanisms. The representation of individuals ranges from bit-strings to real-valued vectors, Lisp expressions, and neural networks. Finally, the relative importance of mutation and crossover (recombination), as well as their particular implementations, differs widely across evolutionary algorithms.

Table 2.4 Fitness evaluation

Individual	CID	Fuel System	Turbo	Valves	Cylinders	Score
$3'$	250	Elect. Inject.	Yes	8	4	250
$4'$	150	2 Barrels	No	12	6	350

Table 2.5 Final population

Individual	CID	Fuel System	Turbo	Valves	Cylinders	Score
3	150	Elect. Inject.	Yes	12	4	300
4	200	2 Barrels	No	8	4	150
$3'$	250	Elect. Inject.	Yes	8	4	250
$4'$	150	2 Barrels	No	12	6	350

2.3 Components of Evolutionary Algorithms

Usually the term evolutionary computation or evolutionary algorithms include the domains of genetic algorithms (GA), evolution strategies, evolutionary programming, and genetic programming. Evolutionary algorithms have a number of components, procedures or operators that must be specified in order to define a particular EA. The most important components of EA are

- Representation
- Evaluation function
- Population
- Parent selection mechanism
- Variation operators, recombination and mutation
- Survivor selection mechanism (Replacement)

Also the initialization procedure and a termination condition must also be defined.

2.4 Representation

Representation is a way of representing solutions/individuals in evolutionary computation methods. This can encode appearance, behavior, physical qualities of individuals. Designing a good representation that is expressive and evolvable is a hard problem in evolutionary computation. Difference in genetic representations in one of the major criteria drawing a line between known classes of evolutionary computation.

Evolutionary algorithm uses linear binary representations. The most standard one is an array of bits. Arrays of other types and structures can be used in essentially the same way. The main property that makes these genetic representations convenient is that their parts are easily aligned due to their fixed size. This facilitates simple crossover operation. Variable length representations were also explored in Genetic algorithms, but crossover implementation is more complex in this case.

Human-based genetic algorithm (HBGA) offers a way to avoid solving hard representation problems by outsourcing all genetic operators to outside agents (in this case, humans). This way the algorithm need not be aware of a particular genetic representation used for any solution. Some of the common genetic representations are

- binary array
- genetic tree
- parse tree
- binary tree
- natural language

2.5 Evaluation/Fitness Function

A fitness function is a particular type of objective function that quantifies the optimality of a solution (that is, a chromosome) in a evolutionary algorithm so that that particular chromosome may be ranked against all the other chromosomes. Optimal chromosomes, or at least chromosomes which are more optimal, are allowed to breed and mix their datasets by any of several techniques, producing a new generation that will (hopefully) be even better.

Another way of looking at fitness functions is in terms of a fitness landscape, which shows the fitness for each possible chromosome. An ideal fitness function correlates closely with the algorithm's goal, and yet may be computed quickly. Speed of execution is very important, as a typical genetic algorithm must be iterated many, many times in order to produce a usable result for a non-trivial problem.

Definition of the fitness function is not straightforward in many cases and often is performed iteratively if the fittest solutions produced by GA are not what are desired. In some cases, it is very hard or impossible to come up even with a guess of what fitness function definition might be. Interactive genetic algorithms address this difficulty by outsourcing evaluation to external agents. Once the genetic representation has been defined, the next step is to associate to each solution (chromosome) a value corresponding to the fitness function. There is generally no problem in determining the fitness function. In fact, most of the time, it is implicitly defined by the problem that is to be optimized. But, as simple as it may look, particular attention should be taken due to the fact the selection is done according to the fitness of individuals. The fitness function should not only indicate how good the solution is, but also it should correspond to how close the chromosome is to the optimal one.

The fitness function interprets the chromosome in terms of physical representation and evaluates its fitness based on traits of being desired in the solution. The fitness function has a higher value when the fitness characteristic of the chromosome is better than others. In addition, the fitness function introduces a criterion for selection of chromosomes.

2.6 Population Initialization

In general, there are two issues to be considered for population initialization of EA: the initial population size and the procedure to initialize population.

It was felt that the population size needed to increase exponentially with the complexity of the problem (i.e., the length of the chromosome) in order to generate good solutions. Recent studies have shown that satisfactory results can be obtained with a much smaller population size. To summarize, a large population is quite useful, but it demands excessive costs in terms of both memory and time. As would be expected, deciding adequate population size is crucial for efficiency.

There are two ways to generate the initial population: heuristic initialization and random initialization. Heuristic initialization explores a small part of the solution space and never finds global optimal solutions because of the lack of diversity in the population. Therefore, random initialization is effected so that the initial population is generated with the encoding method. The algorithm of population initialization is shown below:

```
Procedure: population initialization
    Begin
    Select the first job into the first position
         randomly;
    For i = 0 to code_size(n) //n: length of chromosome
         For j = 0 to job_size(m) //m:number of jobs
              Insert job j that has not inserted into
         the chromosome temporarily;
              Evaluate the fitness value of this
         complete chromosome so far;
              Keep the fitness value in memory;
         End For
    Evaluate the hest fitness value from the memory
    End For
```

2.7 Selection

Selection determines which individuals are chosen for mating (recombination) and how many offspring each selected individual produces. The first step is fitness assignment by:

- rank-based fitness assignment
- multi-objective ranking

Each individual in the selection pool receives a reproduction probability depending on the own objective value and the objective value of all other individuals in the selection pool. This fitness is used for the actual selection step afterwards.
Throughout this section some terms are used for comparing the different selection schemes. The definition of these terms is given below:

- *selective pressure* - probability of the best individual being selected compared to the average probability of selection of all individuals
- *bias* - absolute difference between an individual's normalized fitness and its expected probability of reproduction
- *spread* - range of possible values for the number of offspring of an individual
- *loss of diversity* - proportion of individuals of a population that is not selected during the selection phase

- *selection intensity* - expected average fitness value of the population after applying a selection method to the normalized Gaussian distribution
- *selection variance* - expected variance of the fitness distribution of the population after applying a selection method to the normalized Gaussian distribution

2.7.1 Rank Based Fitness Assignment

In rank-based fitness assignment, the population is sorted according to the objective values. The fitness assigned to each individual depends only on its position in the individuals rank and not on the actual objective value. Rank-based fitness assignment overcomes the scaling problems of the proportional fitness assignment. The reproductive range is limited, so that no individuals generate an excessive number of offspring. Ranking introduces a uniform scaling across the population and provides a simple and effective way of controlling selective pressure. Rank-based fitness assignment behaves in a more robust manner than proportional fitness assignment and, thus, is the method of choice. The common types of ranking are Linear and Non-Linear ranking.

Linear Ranking

Consider *Nind* the number of individuals in the population, *Pos* the position of an individual in this population (least fit individual has $Pos = 1$, the fittest individual $Pos = Nind$) and *SP* the selective pressure. The fitness value for an individual is calculated as:

Linear ranking fitness:

$$Fitness(Pos) = 2 - SP + 2\,^{*}(SP-1)\,^{*}\frac{(Pos-1)}{(Nind-1)} \tag{2.1}$$

Linear ranking allows values of selective pressure in [1.0, 2.0].

Non-linear Ranking

The use of non-linear ranking permits higher selective pressures than the linear ranking method.

Non-linear ranking fitness:

$$Fitness(Pos) = \frac{Nind\,^{*}X^{Pos-1}}{\sum_{i=1}^{Nind} X^{i-1}} \tag{2.2}$$

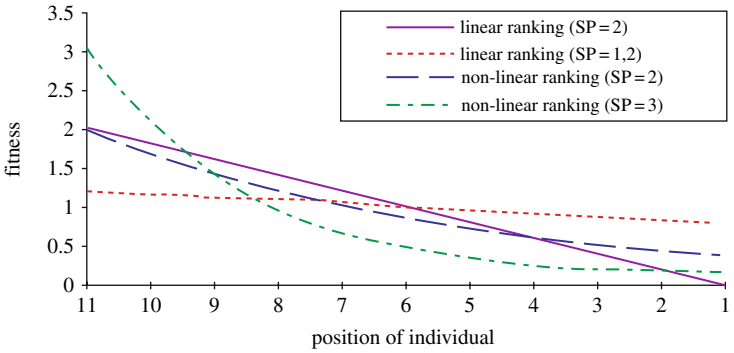


Fig. 2.3 Fitness assignment for linear and non-linear ranking

X is computed as the root of the polynomial:

$$0 = (SP - Nind)\ {}^{*}X^{Nind-1} + SP\ {}^{*}X^{Nind-2} + \ldots\ldots\ldots + SP.X + SP \tag{2.3}$$

Non-linear ranking allows values of selective pressure in [1, *Nind* - 2].

Comparison of Linear and Non-linear Ranking

Figure 2.3 compares linear and non-linear ranking graphically.

The probability of each individual being selected for mating depends on its fitness normalized by the total fitness of the population. Table 2.6 contains the fitness

Table 2.6 Dependency of fitness value from selective pressure

Objective value	Fitness value (parameter: selective pressure)				
	Linear ranking		No ranking	Non-linear ranking	
	2.0	1.1	1.0	3.0	2.0
1	2.0	1.10	1,0	3.00	2.00
3	1.8	1.08	1,0	2.21	1.69
4	1.6	1.06	1,0	1.62	1.43
7	1.4	1.04	1,0	1.99	1.21
8	1.2	1.02	1,0	0.88	1.03
9	1.0	1.00	1,0	0.65	0.87
10	0.8	0.98	1,0	0.48	0.74
15	0.6	0.96	1,0	0.35	0.62
20	0.4	0.94	1,0	0.26	0.53
30	0.2	0.92	1,0	0.19	0.45
95	0.0	0.90	1,0	0.14	0.38

values of the individuals for various values of the selective pressure assuming a population of 11 individuals and a minimization problem.

Analysis of Linear Ranking

An analysis of linear ranking selection can be found. The properties of linear ranking shown in Figure 2.4 are
Selection intensity:

$$SelInt_{LinRank}(SP) = (SP-1).\frac{1}{sqrt(\pi)} \tag{2.4}$$

Loss of diversity:

$$LossDiv_{LinRank}(SP) = \frac{SP-1}{4} \tag{2.5}$$

Selection variance:

$$SelVar_{LinRank}(SP) = 1 - \frac{(SP-1)^2}{\pi} = 1 - SelInt_{LinRank}(SP)^2 \tag{2.6}$$

2.7.2 Multi–objective Ranking

When proportional and rank-based fitness assignment is concerned it is assumed that individuals display only one objective function value. In many real world problems, however, there are several criteria, which have to be considered in order to evaluate the quality of an individual. Only on the basis of the comparison of these

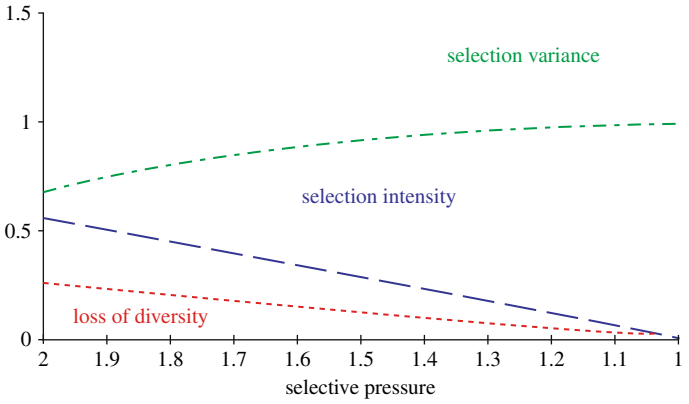


Fig. 2.4 Properties of linear ranking

several criteria (thus multi-objective) can a decision be made as to the superiority of one individual over another. Then, as in single-objective problems, an order of individuals within the population can be established from these reciprocal comparisons - multi-objective ranking. After this order has been established the single-objective ranking methods can be used to convert the order of the individuals to corresponding fitness values.

Multi-objective fitness assignment (and with it *multi-objective optimization*) is concerned with the simultaneous minimization of *NObj* criteria f_r, with $r = 1,\ldots,NObj$. The values f_r are determined by the objective function, which in turn is dependent on the variables of the individuals (the decision variables).

A straightforward example should serve as the motivation for the following considerations. When objects are produced, the production costs should be kept low and the objects should be produced quickly. Various solutions can be developed during production planning which may differ regarding the number and type of the machines employed, as well as regarding the number of workers. The criteria production costs f_1 and production time f_2, both of which are determined by the objective function, serve as evaluation criteria for each solution.

PARETO-ranking

The superiority of one solution over the other can be decided by comparing two solutions. It can be carried out both generally and for as many criteria as desired, following the schema in the following equation.

PARETO-ranking response. PARETO-dominance:

$$\begin{aligned} &\forall i \in \{1,\ldots NObj\}, f_i^{Ldx1} \leq f_i^{Ldx2} and \exists i \in \{1,..NObj\}, f_i^{Ldx1} < f_i^{Ldx2} \\ &\Rightarrow solution1 p < solution2 \\ &(p <: partially\ less\ than\ plt) \end{aligned} \tag{2.7}$$

If solution$_1$ is p< (partially less than) solution$_2$, it follows that solution dominates solution$_2$. In the example used here this means: if costs and time are less for solution$_1$ than for solution$_2$, it follows that solution$_1$ is superior to solution$_2$. It would even be sufficient if one of the two values was equal for both solutions (equal costs) and only the other value was lower (less time).

If, however, none of the solutions dominates the other both solutions are to be regarded as equivalent with respect to the PARETO-order. The same rank is assigned to individuals, which do not dominate each other.

The rank of an individual within the population $(rank_i)$ depends on the number of individuals $(NumInd_{dominated})$ dominating this individual:

$$Rang = 1 + NumInd$$

All solutions that are found during optimization and are not dominated by a different solution constitute the PARETO-optimal solutions (*PARETO-optimal set*) of

this problem (PARETO-optimality). These solutions are assigned a rank value of 1. In the case of each PARETO-optimal solution it is not possible to improve one of the criteria without one or several of the other criteria deteriorating.

PARETO-ranking using a vector of goals for the individual criteria, allows for an improved generation of the sequence of a number of solutions compared to plain PARETO-ranking. Multi-objective optimization is carried out in order to find a number of non-dominated solutions, the PARETO-optimal set. A normal evolutionary algorithm, however, converges at a single solution. This process is termed a genetic drift. For this reason, methods must be built in, which achieve the maintenance or expansion of population diversity (prevention of *premature convergence*).

The actual selection is performed in the next step. Parents are selected according to their fitness by means of one of the following algorithms:

- roulette-wheel selection
- stochastic universal sampling
- local selection
- truncation selection
- tournament selection

2.7.3 Roulette Wheel selection

The simplest selection scheme is roulette-wheel selection, also called stochastic sampling with replacement. This is a stochastic algorithm and involves the following technique: The individuals are mapped to contiguous segments of a line, such that each individual's segment is equal in size to its fitness. A random number is generated and the individual whose segment spans the random number is selected. The process is repeated until the desired number of individuals is obtained (called mating population). This technique is analogous to a roulette wheel with each slice proportional in size to the fitness as shown in Figure 2.5.

Table 2.7 shows the selection probability for 11 individuals, linear ranking and selective pressure of 2 together with the fitness value. Individual 1 is the

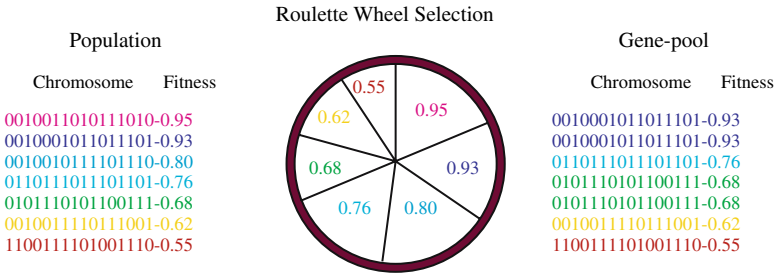


Fig. 2.5 Roulette wheel selection – operation

Table 2.7 Selection probability and fitness value

Number of individual	1	2	3	4	5	6	7	8	9	10	11
Fitness value	2.0	1.8	1.6	1.4	1.2	1.0	0.8	0.6	0.4	0.2	0.0
Selection probability	0.18	0.16	0.15	0.13	0.11	0.09	0.07	0.06	0.03	0.02	0.0

fit individual and occupies the largest interval, whereas individual 10 as the second least fit individual has the smallest interval on the line (see figure 2.5). Individual 11, the least fit interval, has a fitness value of 0 and get no chance for reproduction.

For selecting the mating population the appropriate number of uniformly distributed random numbers (uniform distributed between 0.0 and 1.0) is independently generated.
For a sample of 6 random numbers:
0.81, 0.32, 0.96, 0.01, 0.65, 0.42.
Figure 2.6 shows the selection process of the individuals for the example in table together with the above sample trials.
After selection the mating population consists of the individuals:
1, 2, 3, 5, 6, 9.
The roulette-wheel selection algorithm provides a zero bias but does not guarantee minimum spread.

2.7.4 Stochastic universal sampling

Stochastic universal sampling provides zero bias and minimum spread. The individuals are mapped to contiguous segments of a line, such that each individual's segment is equal in size to its fitness exactly as in roulette-wheel selection. Here equally spaced pointers are placed over the line as many as there are individuals to be selected. Consider *NPointer* the number of individuals to be selected, then the distance between the pointers are 1/*NPointer* and the position of the first pointer is given by a randomly generated number in the range [0, 1/*NPointer*].

For 6 individuals to be selected, the distance between the pointers is $1/6 = 0.167$.
Figure 2.7 shows the selection for the above example.
For a sample of 1 random number in the range [0, 0.167]:0.1.

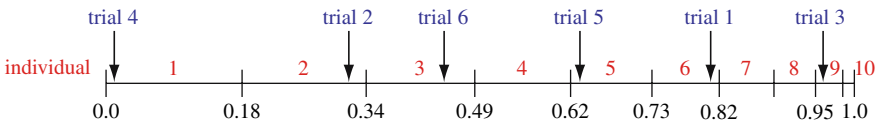


Fig. 2.6 Roulette-wheel selection

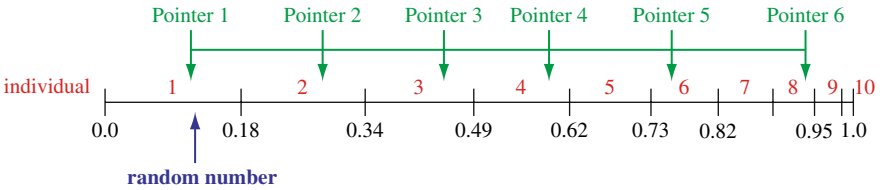


Fig. 2.7 Stochastic universal sampling

After selection the mating population consists of the individuals:
`1, 2, 3, 4, 6, 8.`
Stochastic universal sampling ensures a selection of offspring which is closer to what is deserved then roulette wheel selection.

2.7.5 Local Selection

In local selection every individual resides inside a constrained environment called the local neighborhood. (In the other selection methods the whole population or subpopulation is the selection pool or neighborhood.) Individuals interact only with individuals inside this region. The neighborhood is defined by the structure in which the population is distributed. The neighborhood can be seen as the group of potential mating partners. Local selection is part of the local population model.

The first step is the selection of the first half of the mating population uniform at random (or using one of the other mentioned selection algorithms, for example, stochastic universal sampling or truncation selection). Now a local neighborhood is defined for every selected individual. Inside this neighborhood the mating partner is selected (best, fitness proportional, or uniform at random).

The structure of the neighborhood can be any one of the following types as illustrated in Figures 2.8 and 2.9:

- linear
  - full ring, half ring (see Figure)
- two-dimensional
  - full cross, half cross (see Figure, left)
  - full star, half star (see Figure, right)
- three-dimensional and more complex with any combination of the above structures.

The distance between possible neighbors together with the structure determines the size of the neighborhood. Table 2.8 gives examples for the size of the neighborhood for the given structures and different distance values.

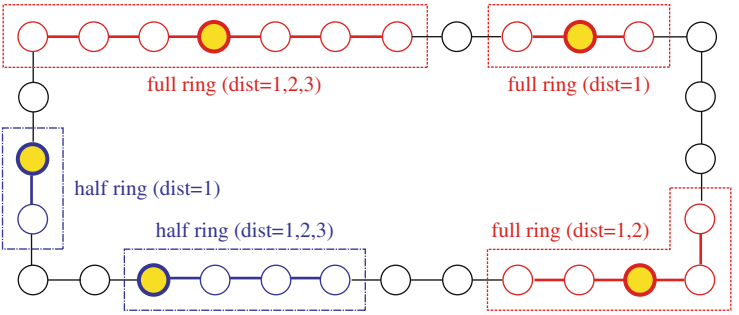


Fig. 2.8 Linear neighborhood: full and half ring

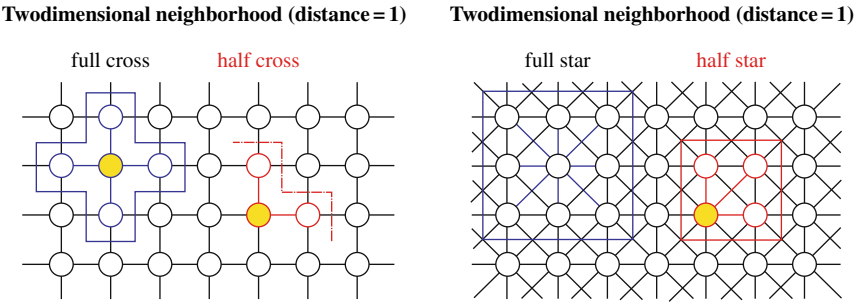


Fig. 2.9 Two-dimensional neighborhood; left: full and half cross, right: full and half star

Between individuals of a population, 'isolation by distance' exists. The smaller the neighborhood, the bigger the isolation distance. However, because of overlapping neighborhoods, propagation of new variants takes place. This assures the exchange of information between all individuals.

The size of the neighborhood determines the speed of propagation of information between the individuals of a population, thus deciding between rapid propagation and maintenance of a high diversity/variability in the population. A higher variability is often desired, thus preventing problems such as premature convergence to a local minimum. Local selection in a small neighborhood performed better than local selection in a bigger neighborhood. Nevertheless, the interconnection

Table 2.8 Number of neighbors for local selection

Structure of selection	Distance	
full ring	2	4
half ring	1	2
full cross	4	8 (12)
half cross	2	4 (5)
full star	8	24
half star	3	8

of the whole population must still be provided. Two-dimensional neighborhood with structure half star using a distance of 1 is recommended for local selection. However, if the population is bigger (>100 individuals) a greater distance and/or another two-dimensional neighborhood should be used.

2.7.6 Truncation Selection

Compared to the previous selection methods that are modeling natural selection, truncation selection is an artificial selection method. It is used by breeders for large populations/mass selection. In truncation selection individuals are sorted according to their fitness. Only the best individuals are selected for parents. These selected parents produce uniform at random offspring. The parameter for truncation selection is the truncation threshold *Trunc*. *Trunc* indicates the proportion of the population to be selected as parents and takes values ranging from 50%–10%. Individuals below the truncation threshold do not produce offspring. The term selection intensity is often used in truncation selection. Table 2.9 shows the relation between both truncation threshold and selection intensity.

Analysis of Truncation Selection

An analysis of truncation selection was performed. The properties of truncation selection shown in Figure 2.10 are:
Selection intensity

$$SelInt_{Truncation}(Trunc) = \frac{1}{Trunc}.\frac{1}{sqrt(2.\pi)}.e^{\frac{t^2}{2}} \tag{2.8}$$

Loss of diversity

$$LossDiv_{Truncation}(Trunc) = 1 - Trunc \tag{2.9}$$

Selection variance

$$SelVar_{Truncation}(Trunc) = 1 - SelInt_{Truncation}(Trunc).(SelInt_{Truncation}(Trunc) - f_c) \tag{2.10}$$

Table 2.9 Relation between truncation threshold and selection intensity

Truncation threshold	1%	10%	20%	40%	50%	80%
Selection intensity	2.66	1.76	1.2	0.97	0.8	0.34

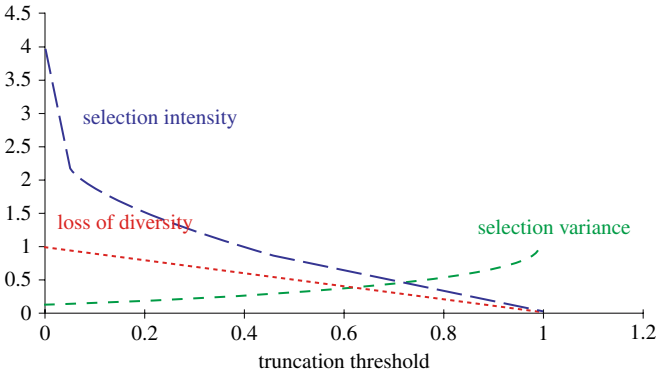


Fig. 2.10 Properties of truncation selection

Tournament Selection

In tournament selection, *Tour* - the number of individuals is chosen randomly from the population and the best individual from this group is selected as parent. This process is repeated as often as individuals must be chosen. These selected parents produce uniform at random offspring. The parameter for tournament selection is the tournament size *Tour. Tour* takes values ranging from 2 to N_{ind} (number of individuals in population). Table 2.10 shows the relation between tournament size and selection intensity.

Analysis of Tournament Selection

An analysis of tournament selection was performed. The properties of tournament selection shown in Figure 2.11 are:
Selection intensity

$$SelInt_{Turnier}(Tour) \approx \sqrt{2 \cdot \left(\ln(Tour) - \ln\left(\sqrt{4.14 \cdot \ln(Tour)}\right)\right)} \tag{2.11}$$

Loss of diversity

$$LossDiv_{Tournament}(Tour) = Tour^{\frac{1}{Tour-1}} - Tour^{\frac{-Tour}{Tour-1}} \tag{2.12}$$

Table 2.10 Relation between tournament size and selection intensity

Tournament size	1	2	3	5	10	30
Selection intensity	0	0.56	0.85	1.15	1.53	2.04

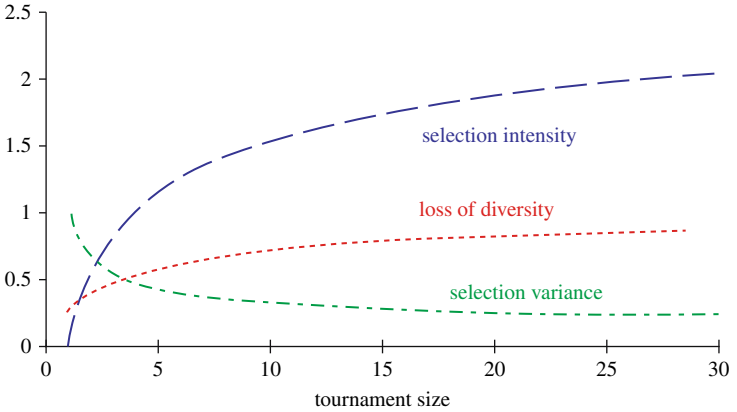


Fig. 2.11 Properties of tournament selection

(About 50% of the population are lost at tournament size Tour = 5).
Selection variance

$$SelVar_{Tournament}(Tour) = \frac{0.918}{\ln(1.186 + 1.328 * Tour)} \tag{2.13}$$

2.7.7 Comparison of Selection Properties

As shown in the previous sections of this chapter the selection methods behave similarly assuming similar selection intensity. This section describes the performance of each selection scheme based on the properties.

Selection Parameter and Selection Intensity

Figure 2.12 shows the relation between selection intensity and the appropriate parameters of the selection methods (selective pressure, truncation threshold and tournament size). It should be stated that with tournament selection only discrete values can be assigned and linear ranking selection allows only a smaller range for the selection intensity.

However, the behavior of the selection methods is different. Thus, the selection methods will be compared on the parameters loss of diversity (Figure 2.13) and selections variance (Figure 2.14) on selection intensity.

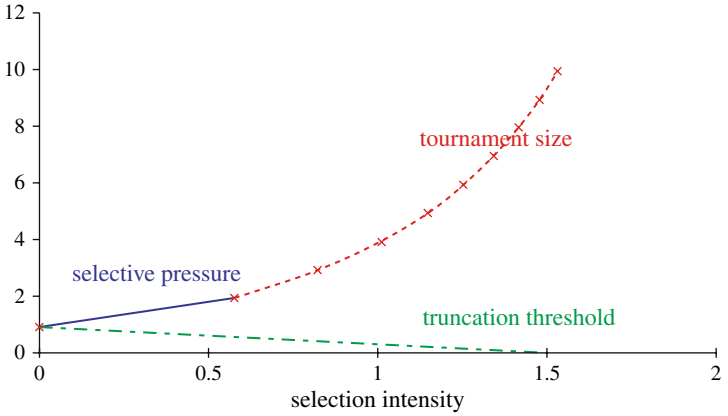


Fig. 2.12 Dependence of selection parameter on selection intensity

Loss of Diversity and Selection Intensity

Truncation selection leads to a much higher loss of diversity for the same selection intensity compared to ranking and tournament selection. Truncation selection is more likely to replace less fit individuals with fitter offspring, because all individuals below a certain fitness threshold do not have a probability to be selected. Ranking and tournament selection seem to behave similarly. However, ranking selection works in an area where tournament selection does not work because of the discrete character of tournament selection.

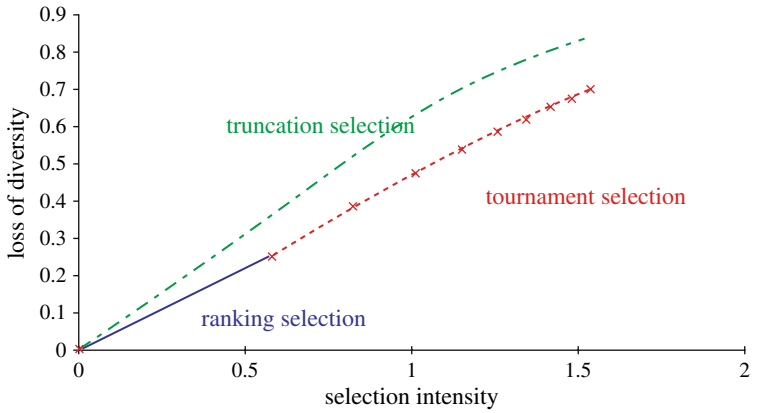


Fig. 2.13 Dependence of loss of diversity on selection intensity

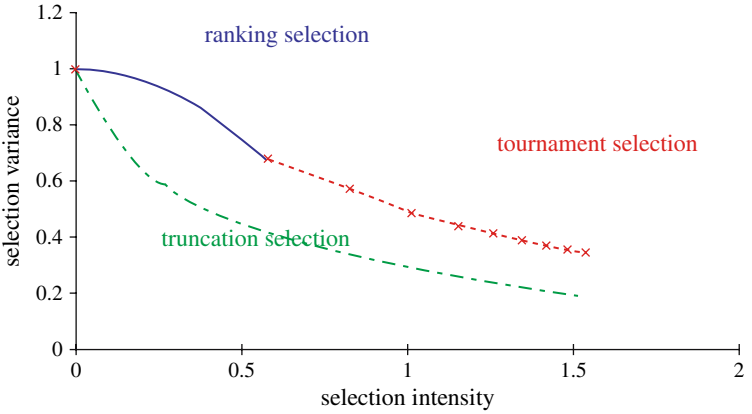


Fig. 2.14 Dependence of selection variance on selection intensity

Selection Variance and Selection Intensity

For the same selection intensity, truncation selection leads to a much smaller selection variance than ranking or tournament selection. As can be seen clearly ranking selection behaves similar to tournament selection. However, again ranking selection works in an area where tournament selection does not work because of the discrete character of tournament selection. It was proven that the fitness distribution for ranking and tournament selection for SP = 2 and Tour = 2 (SelInt = 1/sqrt (pi)) is identical.

2.7.8 MATLAB Code Snippet for Selection

```
function result = select(popWithDistrib)
  % Select some genotypes from the population,
  % and possibly mutates them.
selector = rand;
  total_prob = 0;
  % Default to last in case of rounding error
  genotype = popWithDistrib(end,2:end);
  for i=1:size(popWithDistrib,1)
      total_prob=total_prob+popWithDistrib(i,1);
      if total_prob > selector
        genotype = popWithDistrib(i,2:end);
        break;
      end
  end
  result = mutate(genotype);
```

2.8 Recombination

Recombination produces new individuals in combining the information contained in the parents (parents - mating population). Depending on the representation of the variables of the individuals the types of recombination can be applied:

- All presentation:
  - discrete recombination, (known from recombination of real valued variables), corresponds with uniform crossover, (known from recombination of binary valued variables),
- Real valued recombination
  - intermediate recombination
  - line recombination
  - extended line recombination
- Binary valued recombination
  - single-point/double-point/multi-point crossover
  - uniform crossover
  - shuffle crossover
  - crossover with reduced surrogate
- Other types of crossover
  - Arithmetic crossover
  - Heuristic crossover

For the recombination of binary valued variables the name 'crossover' is established. This has mainly historical reasons. Genetic algorithms mostly used binary variables and the name 'crossover'. Both notions (recombination and crossover) are equivalent in the area of Evolutionary Algorithms. The methods for binary valued variables constitute special cases of the discrete recombination. These methods can all be applied to integer valued and real valued variables as well.

2.8.1 Discrete Recombination

Discrete recombination performs an exchange of variable values between the individuals. For each position the parent who contributes its variable to the offspring is chosen randomly with equal probability.

$$Var_i^O = Var_i^{P_1}.a_i + Var_i^{P_2}.(1-a_i) \quad i \in (1,2,\ldots\ldots N_{\text{var}}), \quad a_i \in \{0,1\}\ uniform\ at\ random, a_i\ for\ each\ i\ defined \tag{2.14}$$

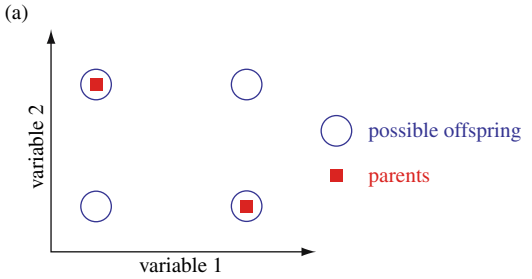


Fig. 2.14a Possible positions of the offspring after discrete recombination

Discrete recombination generates corners of the hypercube defined by the parents. Figure 2.14a shows the geometric effect of discrete recombination.

Consider the following two individuals with 3 variables each (3 dimensions), which will also be used to illustrate the other types of recombination for real valued variables:

```
individual 1     12     25     5
individual 2     123    4      34
```

For each variable the parent who contributes its variable to the offspring is chosen randomly with equal probability:

```
sample 1         2      2      1
sample 2         1      2      1
```

After recombination the new individuals are created:

```
offspring 1      123    4      5
offspring 2      12     4      5
```

Discrete recombination can be used with any kind of variables (binary, integer, real or symbols).

2.8.2 Real Valued Recombination

The recombination methods in this section can be applied for the recombination of individuals with real valued variables.

Fig. 2.15 Area for variable value of offspring compared to parents in intermediate recombination

Intermediate Recombination

Intermediate recombination is a method only applicable to real variables (and not binary variables). Here the variable values of the offspring are chosen somewhere around and between the variable values of the parents.

Offspring are produced according to the rule:

$$Var_i^O = Var_i^P \cdot a_i + Var_i^P \cdot 1 - a_i i \in (1,2,\ldots,N_{var}),$$
$$a_i \in [-d, 1+d] \text{ uniform at random, } d = 0.25, a_i \text{ for each } i \textit{ new} \qquad (2.15)$$

where a is a scaling factor chosen uniformly at random over an interval $[-d,\ 1+d]$ for each variable anew.

The value of the parameter d defines the size of the area for possible offspring. A value of $d = 0$ defines the area for offspring the same size as the area spanned by the parents. This method is called (standard) intermediate recombination. Because most variables of the offspring are not generated on the border of the possible area, the area for the variables shrinks over the generations. This shrinkage occurs just by using (standard) intermediate recombination. This effect can be prevented by using a larger value for d. A value of $d = 0.25$ ensures (statistically), that the variable area of the offspring is the same as the variable area spanned by the variables of the parents. Figure 2.15 represents a picture of the area of the variable range of the offspring defined by the variables of the parents.

Consider the following two individuals with 3 variables each:

```
individual 1     12     25     5
individual 2     123    4      34
```

The chosen a for this example are:

```
sample 1         0.5    1.1    −0.1
sample 2         0.1    0.8    0.5
```

The new individuals are calculated as:

```
offspring 1      67.5   1.9    2.1
offspring 2      23.1   8.2    19.5
```

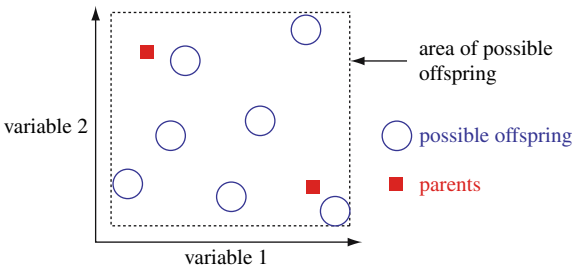


Fig. 2.16 Possible areas of the offspring after intermediate recombination

Intermediate recombination is capable of producing any point within a hypercube slightly larger than that defined by the parents. Figure 2.16 shows the possible area of offspring after intermediate recombination.

Line Recombination

Line recombination is similar to intermediate recombination, except that only one value of a for all variables is used. The same a is used for all variables:

$$Var_i^O = Var_i^P \cdot a_i + Var_i^P \cdot 1 - a_i i \in (1,2,\ldots,N_{var}),$$
$$a_i \in [-d, 1+d] \text{uniform at random},\ d = 0.25, a_i \text{ for all } i \text{ identical} \quad (2.16)$$

For the value of d the statements given for intermediate recombination are applicable.

Consider the following two individuals with 3 variables each:

```
individual 1     12     25     5
individual 2     123    4      34
```

The chosen a for this example are:

```
sample 1         0.5
sample 2         0.1
```

The new individuals are calculated as:

```
offspring 1      67.5   14.5   19.5
offspring 2      23.1   22.9   7.9
```

Line recombination can generate any point on the line defined by the parents. Figure 2.17 shows the possible positions of the offspring after line recombination.

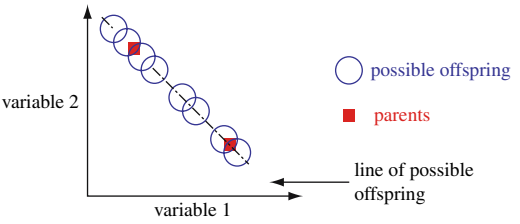


Fig. 2.17 Possible positions of the offspring after line recombination

Extended Line Recombination

Extended line recombination generates offspring on a line defined by the variable values of the parents. However, extended line recombination is not restricted to the line between the parents and a small area outside. The parents just define the line where possible offspring may be created. The size of the area for possible offspring is defined by the domain of the variables.

Inside this possible area the offspring are not uniform at random distributed. The probability of creating offspring near the parents is high. Only with low probability offspring are created far away from the parents. If the fitness of the parents is available, then offspring are more often created in the direction from the worse to the better parent (directed extended line recombination).

Offspring are produced according to the following rule:

$$
\begin{aligned}
Var_i^O = & Var_i^P + s_i \cdot r_i \cdot a_i \cdot \frac{Var_i^P - Var_i^P}{\| Var^P - Var^P \|} i \in (1,2,\ldots,N_{var}), \\
& a_i = 2^{-ku}, k : \text{mutation precision}, u \in [0,1] \text{uniform at random}, \\
& a_i \text{ for all } i \text{ identical}, \\
& r_i = r \cdot domainr : \text{ range of recombination steps}, \\
& s_i \in (-1,+1), \text{ uniform at random} : \text{undirected recombination}, \\
& \quad +1 \text{ with probability} > 0.5 : \text{directedrecombination}, \qquad (2.17)
\end{aligned}
$$

The creation of offspring uses features similar to the mutation operator for real valued variables. The parameter a defines the relative step-size, the parameter r the maximum step-size and the parameter s the direction of the recombination step.

Figure 2.18 tries to visualize the effect of extended line recombination.

The parameter k determines the precision used for the creation of recombination steps. A larger k produces smaller steps. For all values of k the maximum value for a is $a = 1$ $(u = 0)$. The minimum value of a depends on k and is $a = 2^{-k}$ $(u = 1)$. Typical values for the precision parameter k are in the area from 4 to 20.

A robust value for the parameter r (range of recombination step) is 10% of the domain of the variable. However, according to the defined domain of the variables or for special cases this parameter can be adjusted. By selecting a smaller value for r the creation of offspring may be constrained to a smaller area around the parents.

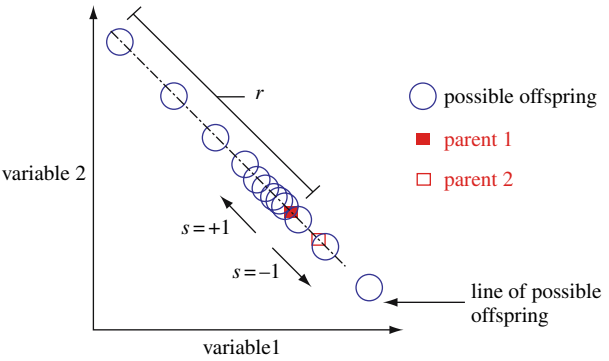


Fig. 2.18 Possible positions of the offspring after extended line recombination according to the positions of the parents and the definition area of the variables

If the parameter s (search direction) is set to -1 or $+1$ with equal probability an undirected recombination takes place. If the probability of $s = +1$ is higher than 0.5, a directed recombination takes place (offspring are created in the direction from the worse to the better parent - the first parent must be the better parent).

Extended line recombination is only applicable to real variables (and not binary or integer variables).

2.8.3 Binary Valued Recombination (Crossover)

This section describes recombination methods for individuals with binary variables. Commonly, these methods are called 'crossover'. During the recombination of binary variables only parts of the individuals are exchanged between the individuals. Depending on the number of parts, the individuals are divided before the exchange of variables (the number of cross points). The number of cross points distinguishes the methods.

Single-point/Double point/Multi-point Crossover

In single-point crossover one crossover position $k\_[1,2,\ldots,Nvar\text{-}1]$, *Nvar*: number of variables of an individual, is selected uniformly at random and the variables exchanged between the individuals about this point, then two new offspring are produced. Figure 2.19 illustrates this process.

Consider the following two individuals with 11 binary variables each:

```
individual 1     0  1  1  1  0  0  1  1  0  1  0
individual 2     1  0  1  0  1  1  0  0  1  0  1
```

The chosen crossover position is:

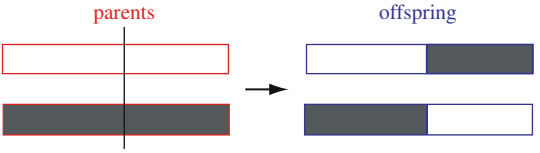


Fig. 2.19 Single-point crossover

```
crossover position          5
```

After crossover the new individuals are created:

```
offspring 1      0  1  1  1  0| 1  0  0  1  0  1
offspring 2      1  0  1  0  1| 0  1  1  0  1  0
```

In double-point crossover two crossover positions are selected uniformly at random and the variables exchanged between the individuals between these points. Then two new offspring are produced.

Single-point and double-point crossover are special cases of the general method multi-point crossover.

For multi-point crossover, m crossover positions k_i-$[1,2,\ldots,Nvar-1]$, $i=1:m$, $Nvar$: number of variables of an individual, are chosen at random with no duplicates and sorted into ascending order. Then, the variables between successive crossover points are exchanged between the two parents to produce two new offspring. The section between the first variable and the first crossover point is not exchanged between individuals. Figure 2.20 illustrates this process.

Consider the following two individuals with 11 binary variables each:

```
individual 1     0  1  1  1  0  0  1  1  0  1  0
individual 2     1  0  1  0  1  1  0  0  1  0  1
```

The chosen crossover positions are:

```
cross pos. (m=3)    2        6        10
```

After crossover the new individuals are created:

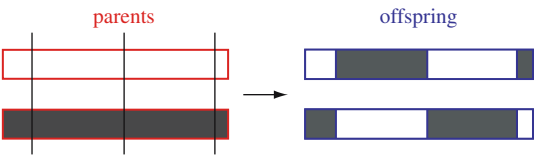


Fig. 2.20 Multi-point crossover

```
offspring 1      0  1| 1  0  1  1| 0  1  1  1| 1
offspring 2      1  0| 1  1  0  0| 0  0  1  0| 0
```

The idea behind multi-point, and indeed many of the variations on the crossover operator, is that parts of the chromosome representation that contribute most to the performance of a particular individual may not necessarily be contained in adjacent substrings. Further, the disruptive nature of multi-point crossover appears to encourage the exploration of the search space, rather than favoring the convergence to highly fit individuals early in the search, thus making the search more robust.

Uniform Crossover

Single and multi-point crossover defines cross points as places between loci where an individual can be split. Uniform crossover generalizes this scheme to make every locus a potential crossover point. A crossover mask, the same length as the individual structure is created at random and the parity of the bits in the mask indicate which parent will supply the offspring with which bits. This method is identical to discrete recombination.

Consider the following two individuals with 11 binary variables each:

```
individual 1     0  1  1  1  0  0  1  1  0  1  0
individual 2     1  0  1  0  1  1  0  0  1  0  1
```

For each variable the parent who contributes its variable to the offspring is chosen randomly with equal probability. Here, the offspring 1 is produced by taking the bit from parent 1 if the corresponding mask bit is 1 or the bit from parent 2 if the corresponding mask bit is 0. Offspring 2 is created using the inverse of the mask, usually.

```
sample 1         0  1  1  0  0  0  1  1  0  1  0
sample 2         1  0  0  1  1  1  0  0  1  0  1
```

After crossover the new individuals are created:

```
offspring 1      1  1  1  0  1  1  1  1  1  1  1
offspring 2      0  0  1  1  0  0  0  0  0  0  0
```

Uniform crossover, like multi-point crossover, has been claimed to reduce the bias associated with the length of the binary representation used and the particular coding for a given parameter set. This helps to overcome the bias in single-point crossover towards short substrings without requiring precise understanding of the significance of the individual bits in the individual representation. Uniform crossover may be parameterized by applying a probability to the swapping of bits. This extra parameter can be used to control the amount of disruption during recombination without introducing a bias towards the length of the representation used.

Shuffle Crossover

Shuffle crossover is related to uniform crossover. A single crossover position (as in single-point crossover) is selected. But before the variables are exchanged, they are randomly shuffled in both parents. After recombination, the variables in the offspring are unshuffled in reverse. This removes positional bias as the variables are randomly reassigned each time crossover is performed.

Crossover with Reduced Surrogate

The reduced surrogate operator constrains crossover to always produce new individuals wherever possible. This is implemented by restricting the location of crossover points such that crossover points only occur where gene values differ.

Arithmetic Crossover

A crossover operator that linearly combines two parent chromosome vectors to produce two new offspring according to the following equations:

$$Offspring1 = \mathrm{a} * Parent1 + (1 - \mathrm{a}) * Parent2$$
$$Offspring2 = (1 - \mathrm{a}) * Parent1 + \mathrm{a} * Parent2$$

where a is a random weighting factor (chosen before each crossover operation). Consider the following 2 parents (each consisting of 4 float genes), which have been selected for crossover:

Parent 1: (0.3) (1.4) (0.2) (7.4)
Parent 2: (0.5) (4.5) (0.1) (5.6)
If $\mathrm{a} = 0.7$, the following two offspring would be produced:
Offspring 1: (0.36) (2.33) (0.17) (6.86)
Offspring 2: (0.402) (2.981) (0.149) (6.842)

Heuristic Crossover

A crossover operator that uses the fitness values of the two parent chromosomes to determine the direction of the search. The offspring are created according to the following equations:

$$Offspring1 = BestParent + \mathrm{r} * (BestParent - WorstParent)$$
$$Offspring2 = BestParent$$

where r is a random number between 0 and 1.

It is possible that offspring1 will not be feasible. This can happen if r is chosen such that one or more of its genes fall outside of the allowable upper or lower bounds. For this reason, heuristic crossover has a user settable parameter (n) for the number of times to try and find an r that result in a feasible chromosome. If a feasible chromosome is not produced after n tries, the WorstParent is returned as Offspring1.

2.8.3.1 MATLAB Code Snippet for Crossover

```
function [x,y] = crossover(x,y)
% Possibly takes some information from one genotype and
% swaps it with information from another genotype

if rand < 0.6
     gene_length = size(x,2);
     % site is between 2 and gene_length
     site = ceil(rand * (gene_length--1)) + 1;
     tmp = x(site:gene_length);
     x(site:gene_length) = y(site:gene_length);
     y(site:gene_length) = tmp;
  end
```

2.9 Mutation

After recombination every offspring undergoes mutation. Offspring variables are mutated by small perturbations (size of the mutation step), with low probability. The representation of the variables determines the used algorithm. Two operators explained here are:

- mutation operator for real valued variables
- mutation for binary valued variables

By mutation individuals are randomly altered. These variations (mutation steps) are mostly small. They will be applied to the variables of the individuals with a low probability (mutation probability or mutation rate). Normally, offspring are mutated after being created by recombination.
For the definition of the mutation steps and the mutation rate two approaches exist:

- Both parameters are constant during a whole evolutionary run. Examples are methods for the mutation of real variables, and mutation of binary variables.
- One or both parameters are adapted according to previous mutations. Examples are the methods for the adaptation of mutation step-sizes known from the area of evolutionary strategies.

2.9.1 Real Valued Mutation

Mutation of real variables means, that randomly created values are added to the variables with a low probability. Thus, the probability of mutating a variable (mutation rate) and the size of the changes for each mutated variable (mutation step) must be defined.

The probability of mutating a variable is inversely proportional to the number of variables (dimensions). The more dimensions one individual has, the smaller is the mutation probability. Different papers reported results for the optimal mutation rate. A mutation rate of $1/n$ (n: number of variables of an individual) produced good results for a wide variety of test functions. That means that per mutation only one variable per individual is changed/mutated. Thus, the mutation rate is independent of the size of the population.

For unimodal functions a mutation rate of $1/n$ was the best choice. An increase in the mutation rate at the beginning connected with a decrease in the mutation rate to $1/n$ at the end gave only an insignificant acceleration of the search.

The given recommendations for the mutation rate are only correct for separable functions. However, most real world functions are not fully separable. For these functions no recommendations for the mutation rate can be given. As long as nothing else is known, a mutation rate of $1/n$ is suggested as well.

The size of the mutation step is usually difficult to choose. The optimal step-size depends on the problem considered and may even vary during the optimization process. It is known, that small steps (small mutation steps) are often successful, especially when the individual is already well adapted. However, larger changes (large mutation steps) can, when successful, produce good results much quicker. Thus, a good mutation operator should often produce small step-sizes with a high probability and large step-sizes with a low probability.

Mutation operator of the *Breeder Genetic Algorithm* is:

$$\begin{aligned} Var_i^{Mut} = & Var_i + s_i \cdot r_i \cdot a_i i \in (1,2,\ldots,n) \text{ uniform at random,} \\ & s_i \in (-1,+1) \text{ uniform at random} \\ & r_i = r \cdot domain_i, r : \text{mutation range (standard:10\%),} \\ & a_i = 2^{-u \cdot k}, u \in [0,1] \text{ uniform at random, } k: \text{ mutation precision,} \end{aligned} \quad (2.18)$$

This mutation algorithm is able to generate most points in the hyper-cube defined by the variables of the individual and range of the mutation (the range of mutation is given by the value of the parameter r and the domain of the variables). Most mutated individuals will be generated near the individual before mutation. Only some mutated individuals will be far away from the not mutated individual. That means, the probability of small step-sizes is greater than that of bigger steps. Figure 2.21 tries to give an impression of the mutation results of mutation operator.

The parameter k (mutation precision) defines indirectly the minimal step-size possible and the distribution of mutation steps inside the mutation range. The

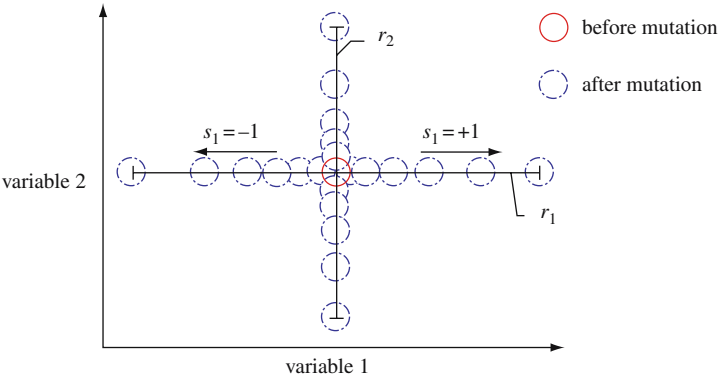


Fig. 2.21 Effect of mutation of real variables in two dimensions

smallest relative mutation step-size is 2^{-k}, the largest $2^0 = 1$. Thus, the mutation steps are created inside the area $[r,\ r \cdot 2^{-k}]$ (mutation range).

With a mutation precision of $k = 16$, the smallest mutation step possible is $r.\ 2^{-16}$. Thus, when the variables of an individual are so close to the optimum, a further improvement is not possible. This can be circumvented by decreasing the mutation range (restart of the evolutionary run or use of multiple strategies)

Typical values for the parameters of the mutation operator from the above equation are:

$$\text{Mutation precision } k : k\ \varepsilon\{4,5,\ldots\ldots,20\}$$

$$\text{Mutation Range } r : r\ \varepsilon[0.1,\ 10^{-6}]$$

By changing these parameters different search strategies can be defined.

2.9.2 Binary mutation

For binary valued individuals mutation means the flipping of variable values, because every variable has only two states. Thus, the size of the mutation step is always 1. For every individual the variable value to change is chosen (mostly uniform at random). Table 2.10 shows an example of a binary mutation for an individual with 11 variables, where variable 4 is mutated.

Table 2.10 Individual before and after binary mutation

Before mutation	0	1	1	*1*	0	0	1	1	0	1	0
				⇓							
After mutation	0	1	1	*0*	0	0	1	1	0	1	0

Table 2.11 Result of the binary mutation

Scaling	Linear		Logarithmic	
Coding	Binary	Gray	Binary	Gray
Before mutation	5.0537	4.2887	2.8211	2.3196
After mutation	4.4910	3.3346	2.4428	1.8172

Assuming that the above individual decodes a real number in the bounds [1, 10], the effect of the mutation depends on the actual coding. Table 2.11 shows the different numbers of the individual before and after mutation for binary/gray and arithmetic/logarithmic coding.

However, there is no longer a reason to decode real variables into binary variables.

2.9.3 Real Valued Mutation with Adaptation of Step Sizes

For the mutation of real variables there is a possibility to learn the direction and step-size of successful mutations by adapting these values. These methods are a part of evolutionary strategies and evolutionary programming. For storing the additional mutation step-sizes and directions additional variables are added to every individual. The number of these additional variables depends on the number of variables n and the method. Each step-size corresponds to one additional variable, one direction to n additional variables. To store n directions n^2 additional variables would be needed.

In addition, for the adaptation of n step-sizes n generations with the calculation of multiple individuals each are needed. With n step-sizes and one direction this adaptation takes $2n$ generations, for n directions n^2 generations. When looking at the additional storage space required and the time needed for adaptation it can be derived, that only the first two methods are useful for practical application. Only these methods achieve an adaptation with acceptable expenditure. The adaptation of n directions is currently only applicable to small problems.

The algorithms for these mutation operators will not be described at this stage. Instead, the interested reader will be directed towards the publications mentioned. Some comments important for the practical use of these operators will be given in the following paragraphs. The mutation operators with step-size adaptation need a different setup for the evolutionary algorithm parameters compared to the other algorithms. The adapting operators employ a small population. Each of these individuals produces a large number of offspring. Only the best of the offspring are reinserted into the population. All parents will be replaced. The selection pressure is 1, because all individuals produce the same number of offspring. No recombination takes place.

Best values for the mentioned parameters are:

- 1 (1–3) individuals per population or subpopulation,
- 5 (3–10) offspring per individual$=>$ generation gap $= 5$,

- the best offspring replace parents=> reinsertion rate = 1,
- no selection pressure=> $SP = 1$,
- no recombination.

When these mutation operators were used one problem had to be solved: the initial size of the individual step-sizes. The original publications just give a value of 1. This value is only suitable for a limited number of artificial test functions and when the domain of all variables is equal. For practical use the individual initial step-sizes must be defined depending on the domain of each variable. Further, a problem-specific scaling of the initial step-sizes should be possible. To achieve this, the parameter mutation range r can be used, similar to the real valued mutation operator.

Typical values for the mutation range of the adapting mutation operators are:

$$\text{Mutation range } r : r\ \varepsilon[10^{-3},\ 10^{-7}]$$

The mutation range determines the initialization of the step-sizes at the beginning of a run only. During the following step-size adaptation the step-sizes are not constrained.

A larger value for the mutation range produces larger initial mutation steps. The offspring are created far away from the parents. Thus, a rough search is performed at the beginning of a run. A small value for the mutation range determines a detailed search at the beginning. Between both extremes the best way to solve the problem at hand must be selected. If the search is too rough, no adaptation takes place. If the initial step sizes are too small, the search takes extraordinarily long and/or the search gets stuck in the next small local minimum.

The adapting mutation operators should be especially powerful for the solution of problems with correlated variables. By the adaptation of step-sizes and directions the correlations between variables can be learned. Some problems (for instance the Rosenbrock function - contains a small and curve shaped valley) can be solved very effectively by adapting mutation operators.

The use of the adapting mutation operators is very difficult (or useless), when the objective function contains many minima (extrema) or is noisy.

2.9.4 Advanced Mutation

In crossover techniques, over-population of these fit members gathers and converges to decrease the chances that crossover will operate again. Therefore, the call for a different technique to handle the new case is necessary. Mutation thrown into the mix is the technique used to hunt for better members among the existing elite members. This operation will increase the fitness's binary mutation while the algorithm is operating. Mutation focuses on similarities between the parents of a given child; the more similarities there are the higher the chance for a mutation. This process cuts down the probability of having premature convergence.

Reproduction, crossover, and mutation operators are the main commonly used operators in genetic algorithms among others. In the reproduction process, only superior strings survive by making several copies of their genetic characteristics. There are some reproduction techniques such as the tournament selection. In the crossover process, two cross sections of parent strings with good fitness values are exchanged to form new chromosomes. Conversely, in the mutation process, only parts of a given string are altered in the hope of obtaining genetically a better child. These operators limit the chances of inferior strings to survive in the successive generations. Moreover, if superior strings existed or are created during the process, they will likely make it to the subsequent generations.

2.9.5 Other Types of Mutation

Flip Bit

A mutation operator that simply inverts the value of the chosen gene (0 goes to 1 and 1 goes to 0). This mutation operator can only be used for binary genes.

Boundary

A mutation operator that replaces the value of the chosen gene with either the upper or lower bound for that gene (chosen randomly). This mutation operator can only be used for integer and float genes.

Non-uniform

A mutation operator that increases the probability that the amount of the mutation will be close to 0 as the generation number increases. This mutation operator keeps the population from stagnating in the early stages of the evolution then allows the genetic algorithm to fine tune the solution in the later stages of evolution. This mutation operator can only be used for integer and float genes.

Uniform

A mutation operator that replaces the value of the chosen gene with a uniform random value selected between the user-specified upper and lower bounds for that gene. This mutation operator can only be used for integer and float genes.

Gaussian

A mutation operator that adds a unit Gaussian distributed random value to the chosen gene. The new gene value is clipped if it falls outside of the user-specified lower or upper bounds for that gene. This mutation operator can only be used for integer and float genes.

2.9.6 MATLAB Code Snippet for Mutation

```
function result=mutate(genotype)
% Possibly mutates a genotype
result=abs(genotype - (rand(size(genotype,1),size
     (genotype,2))<0.03));
```

2.10 Reinsertion

Once the offspring have been produced by selection, recombination and mutation of individuals from the old population, the fitness of the offspring may be determined. If less offspring are produced than the size of the original population then to maintain the size of the original population, the offspring have to be reinserted into the old population. Similarly, if not all offspring are to be used at each generation or if more offspring are generated than the size of the old population then a reinsertion scheme must be used to determine which individuals are to exist in the new population.

The used selection algorithm determines the reinsertion scheme:

- global reinsertion for all population based selection algorithm (roulette-wheel selection, stochastic universal sampling, truncation selection),
- local reinsertion for local selection.

2.10.1 Global Reinsertion

Different schemes of global reinsertion exist:

- Produce as many offspring as parents and replace all parents by the offspring (pure reinsertion).
- Produce less offspring than parents and replace parents uniformly at random (uniform reinsertion).
- Produce less offspring than parents and replace the worst parents (elitist reinsertion).
- Produce more offspring than needed for reinsertion and reinsert only the best offspring (fitness-based reinsertion).

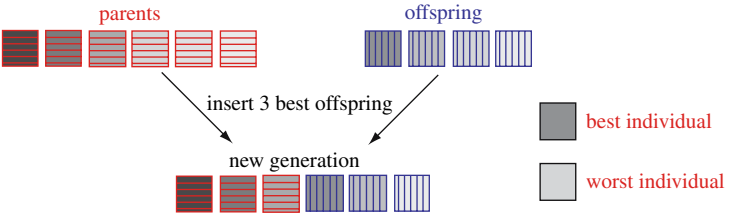


Fig. 2.22 Scheme for elitist insertion

Pure Reinsertion is the simplest reinsertion scheme. Every individual lives one generation only. This scheme is used in the simple genetic algorithm. However, it is very likely, that very good individuals are replaced without producing better offspring and thus, good information is lost.

The elitist combined with fitness-based reinsertion prevents this losing of information and is the recommended method. At each generation, a given number of the least fit parents are replaced by the same number of the most fit offsprings as shown in Figure 2.22. The fitness-based reinsertion scheme implements a truncation selection between offspring before inserting them into the population (i.e. before they can participate in the reproduction process). On the other hand, the best individuals can live for many generations. However, with every generation some new individuals are inserted. It is determined whether the parents are replaced by better or worse offspring.

Because parents may be replaced by offspring with a lower fitness, the average fitness of the population can decrease. However, if the inserted offspring are extremely bad, they will be replaced with new offspring in the next generation.

2.10.2 Local Reinsertion

In local selection individuals are selected in a bounded neighborhood. The reinsertion of offspring takes place in exactly the same neighborhood. Thus, the locality of the information is preserved. The used neighborhood structures are the same as in local selection. The parent of an individual is the first selected parent in this neighborhood. For the selection of parents to be replaced and for selection of offspring to reinsert the following schemes are possible:

- insert every offspring and replace individuals in neighborhood uniformly at random,
- insert every offspring and replace weakest individuals in neighborhood,
- insert offspring fitter than weakest individual in neighborhood and replace weakest individuals in neighborhood,
- insert offspring fitter than weakest individual in neighborhood and replace parent,
- insert offspring fitter than weakest individual in neighborhood and replace individuals in neighborhood uniformly at random,
- insert offspring fitter than parent and replace parent.

2.11 Reproduction Operator

The reproduction operator functions by considering promising offspring "strings" to be potential candidates for reproduction and discarding the weak ones without changing the range of the population. Therefore, the reproduction operator operates in the following manner:

(i) Among the diverse strings in the population, the operator detects only potential candidates that qualify for reproduction.
(ii) The operator then creates manifold versions of these candidates.
(iii) It replaces the superior candidates for the potentially less or inferior candidates.

Reproduction preserves the size of the population. There are some frequently used techniques in identifying good candidates for reproduction, which include *rank grouping, proportionate grouping*, and *tournament grouping*. The purpose of ranking is to prevent rapid convergence.

Rank selection involves individuals in a particular population to be positioned according to their expected value and fitness, but not absolute fitness. The main reason for not scaling the fitness is due to the fact that the absolute fitness is often times masked. Hence, there are pros and cons to this issue. One advantage to this removal of the absolute fitness is when using it, one may obtain convergence problems, while a disadvantage may be vital to identify that an individual is far better than its closest competitor. In addition, positioning eludes in providing the farthest and greatest distribution of the progeny to a smaller cluster of extremely qualified individuals. Therefore, this diminishes the selection burden upon an excessive fitness variance. Moreover, it furthers the selection burden upon a minimal fitness variance. Further, it does not matter if the absolute fitness is elevated or dilapidated, the ratio of the expected value will remain to be the equivalent. The ranking technique (i.e. the fitness proportionate selection) necessitates two elapses for each origination. One of the elapses is when a computation of the expected value of the fitness is preformed for each individual. In addition, it lapses to perform the mean fitness. This technique necessitates the arranging of full population by order. This is an extremely time-consuming technique. The technique of tournament selection is very much like ranking. The advantage of using this technique is that it is more effective in parallel implementation.

Tournament selection does not consume much time, it is much more efficient than the ranking method with regards to the time computation and it is more acquiescent. A selection of two individuals is made from the indiscriminate population. An unbiased number, n is selected from population, such that n is anywhere between zero and one, and m is a parameter. The better of the two selections is chosen to be the parent and it is defined by $n < m$. If this is not the case, then the reduced amount of fit of the individual is chosen. Subsequently, the two individuals are taken back to the initial population, so they may be reselected. In tournament grouping, two strings compete for a spot in the new generation and off course, the winner string superior candidate takes the spot. The process is repeated again with another two strings to fill another spot under the condition that each string may

not participate in more than two tournaments. In this manner, a superior string will have the opportunity to enter the competition twice and thus guarantee two spots. In the same fashion, an inferior string will also have the chance to compete and loose twice. Therefore, every string will either have 0, 1, or 2 representations in the new generation. It has been shown that tournament grouping for replication (reproduction) operator conjoins at a highly rapid pace and needs less processes than the other operator.

Illustration

Consider a sample population of seven strings "containers" with corresponding fitness values "cost" of $\{23, 26, 29, 31, 38, 41, 42\}$. Suppose that 23, 29, 31, 38 and 42 enter the tournament. The rule here is that every string has two chances to compete, also if say 29 and 31 compete; 29 occupies the spot in the new generation because the objective is to minimize the cost. Here is a random tournament. Without loss of generality, let Φ represent the reproduction operator:

$29\Phi 31 \rightarrow 29,\ 41\Phi 42 \rightarrow 41,\ 38\Phi 23 \rightarrow 23,\ 29\Phi 38 \rightarrow 29,\ 23\Phi 26 \rightarrow 23,\ 26\Phi 42 \rightarrow 26,\ 31\Phi 38 \rightarrow 31$. Hence, the winner of this tournament that generates the breeding pool consists of the following set of strings $\{23, 23, 26, 29, 29, 31, 41\}$.

2.11.1 MATLAB Code Snippet for Reproduction

```
function result = reproduce(population)
% Returns next generation of a population

  fitted_pop = distribution(population);
  for i = 1:(size(population,1)/2)
     x = select(fitted_pop);
     y = select(fitted_pop);
     [x,y] = crossover(x,y);
     result(2*i-1,:) = x;
     result(2*i,:) = y;
  end
```

2.12 Categorization of Parallel Evolutionary Algorithms

During recent years the area of Evolutionary Algorithms in general and the field of Parallel Evolutionary Algorithms (PEAs) in particular has matured up to a point, where the application to a complex real-world problem in some applied science

is now potentially feasible and to the benefit of both fields. The effectiveness of EAs, is limited by their ability to balance the need for a diverse set of sampling points with the desire to quickly focus search upon potential solutions. Due to increasing demands such as searching large search spaces with costly evaluation functions and using large population sizes, there is an ever-growing need for fast implementations to allow quick and flexible experimentation. Most EAs work with one large panmictic population. Those EAs suffer from the problem that natural selection relies on the fitness distribution over the whole population. Parallel processing is the natural route to explore. Parallel and distributed computing is a key technology in the present days of networked and high-performance systems. The goal of increased performance can be met in principle by adding processors, memory and an interconnection network and putting them to work together on a given problem. By sharing the workload, it is hoped that an N-processor system will give rise to a speedup in the computation time. Speedup is defined as the time it takes a single processor to execute a given problem instance with a given algorithm divided by the time on a N-processor architecture of the same type for the same problem instance and with the same algorithm. Sometimes, a different, more suitable algorithm is used in the parallel case. Clearly, in the ideal case the maximum speedup is equal to N. If speedup is indeed about linear in the number of processors, then time-consuming problems can be solved in parallel in a fraction of the uniprocessor time or larger and more interesting problem instances can be tackled in the same amount of time. In reality things are not so simple since in most cases several overhead factors contribute to significantly lower the theoretical performance improvement expectations. Furthermore, general parallel programming models turn out to be difficult to design due to the large architectural space that they must span and to the resistance represented by current programming paradigms and languages. In any event, many important problems are sufficiently regular in their space and time dimensions as to be suitable for parallel or distributed computing and evolutionary algorithms are certainly among those.

Furthermore, some of the difficulties that face standard EAs (such as premature convergence, and searching multimodal spaces) may be less of a problem for parallel variants. The two approaches to PEAs are the *standard parallel* approach, using the PEA as a means of implementing a sequential or parallel EA, and the *decomposition* approach with the PEA as a particular model of an EA.

In the first approach, the sequential EA model is implemented on a parallel computer. This is usually done by dividing the task of evaluating the population among several processors. In a PEA model, the full population exists in distributed form. Either multiple independent or interacting subpopulations exist (*coarse-grained* or *distributed* EA), or there is only one population with each population member interacting only with a limited set of neighbors (*fine-grained* EA). The interaction between populations, or members of a population, takes place with respect to a spatial structure of the population. The PEAs are classified according to this spatial structure, the granularity of the distributed population, and the manner in which the EA operators are applied. In a coarse-grained PEA, the population is divided into several (usually equal) subpopulations, each of which runs an EA independently

and in parallel on its own subpopulation. Occasionally, fit individuals migrate randomly from one subpopulation to another (*island model*). In some implementations migrant individuals may move only to geographically nearby subpopulations (islands), rather than to any arbitrary subpopulation (*stepping-stone model*).

In a fine-grained PEA, the population is divided so that each individual is assigned to one processor. Individuals select from, crossover with, and replace only individuals in a bounded region (neighborhood/deme). Since neighborhoods overlap, fit individuals will propagate through the whole population (*Diffusion or isolation-by-distance or neighborhood model*).

The final method to parallelize EAs uses some combination of the previous methods, with some added complexity in some cases. These models maintain more diverse subpopulations mitigating the problem of premature convergence. They also naturally fit the model of the way evolution is viewed as occurring, with a large degree of independence in the global population. Parallel EAs based on Subpopulation Modeling can even be considered as creating new paradigms within this area and thus establishing a new and promising field of research.

2.13 Advantages of Evolutionary Algorithms

The advantages of using evolutionary algorithms over the other global optimization techniques are

1 The performance of evolutionary algorithm is representation independent in contrast to other numerical techniques, which might be applicable for only continuous values or other constrained sets.
2 Evolutionary algorithms offer a framework such that it is comparably easy to incorporate prior knowledge about the problem. Incorporating such information focuses the evolutionary search, yielding a more efficient exploration of the state space of possible solutions.
3 Evolutionary algorithms can also be combined with more traditional optimization techniques. This may be as simple as the use of a gradient minimization after primary search with an evolutionary algorithm (e.g. fine tuning of weights of an evolutionary neural network) or it may involve simultaneous application of other algorithms (e.g. hybridizing with simulated annealing or Tabu search to improve the efficiency of basic evolutionary search).
4 The evaluation of each solution can be handled in parallel and only selection (which requires at least pair-wise competition) requires some serial processing. Implicit parallelism is not possible in many global optimization algorithms like simulated annealing and Tabu search.
5 Traditional methods of optimization are not robust to the dynamic changes in the problem of the environment and often require a complete restart in order to provide a solution (e.g. dynamic programming). In contrast, evolutionary algorithms can be used to adapt solutions to changing circumstance.

6 Perhaps, the greatest advantage of evolutionary algorithms comes from the ability to address problems for which there are no human experts. Although human expertise should be used when it is available, it often proves less than adequate for automating problem-solving routines.

2.14 Multi-objective Evolutionary Algorithms

When there are many (possibly conflicting) objectives to be optimized simultaneously, there can no longer be a single optimal solution but rather a whole set of possible solutions of equivalent quality. Consider, for example, the design of an automobile. Possible objectives could be: minimize cost, maximize speed, minimize fuel consumption and maximize luxury. These goals are clearly conflicting and, therefore, there is no single optimum to be found. Multi-objective EAs can yield a whole set of potential solutions – which are all optimal in some sense - and give the engineers the option to assess the trade-offs between different designs. One then could, for example, choose to create three different cars according to different marketing needs: a slow low-cost model which consumes least fuel, an intermediate solution, and a luxury sports car where speed is clearly the prime objective. Evolutionary algorithms are well suited to multi-objective optimization problems as they are fundamentally based on biological processes which are inherently multi-objective. Figure 2.23 shows the pseudo code of an EA for single and multi-objective optimization.

Evolutionary algorithms seem to be especially suited to multi-objective optimization because they are able to capture multiple Pareto-optimal solutions in a single run, and may exploit similarities of solutions by recombination. Indeed, some research suggests that in MOPs, it is necessary to find not one but several solutions, in order to determine the entire Pareto front. Nevertheless, due to stochastic errors associated with the evolutionary operators, EAs can converge to a single solution. There exist several methods in literature, called *niching techniques* to preserve diversity in the population, in order to converge to different solutions. These techniques can also be applied to MOP.

Task1: Initiate population
Repeat $t = 1,2,\ldots$
Task2: Evaluate solutions in the population:
a) for each individual obtain a scalar fitness
Task3: Perform competitive selection in population
Task4: Apply variation operators to the population
Until *convergence criterion is satisfied*

Fig. 2.23a. Pseudo-code of an EA for single-objective optimization

Task 1: Initiate population
Repeat $t = 1,2\ldots$
Task 2: Evaluate solutions in the population:

a) for each individual obtain a vector fitness
b) for each individual convert vector fitness into a scalar fitness
Task3: Perform competitive selection in the population
Task4: Apply variation operators to the population
***U**ntil convergence criterion is satisfied*

Fig. 2.23b. Pseudo-code of an EA for multiobjective optimization

2.15 Critical Issues in Designing an Evolutionary Algorithm

There are some issues that should be kept in mind when designing and running an evolutionary algorithm. One crucial issue when running an EA is to try to preserve the genetic diversity of the population as long as possible. Opposite to many other optimization methods, EAs use a whole population of individuals – and this is one of the reasons for their power. However, if that populations starts to concentrate in a very narrow region of the search space, all advantages of handling many different individuals vanish, while the burden of computing their fitnesses remains. This phenomenon is known as premature convergence. There are two main directions to prevent this:

- A priori ensuring creation of new material, for instance by using a high level of mutation;
- A posteriori manipulating the fitnesses of all individuals to create a bias against being similar, or close to, existing candidates.

A well-known technique is the so-called niching mechanism. Exploration and exploitation are two terms often used in EC. Although crisp definitions are lacking there has been a lot of discussion about them. The dilemma within an optimization procedure is whether to search around the best-so-far solutions (as their neighborhood hopefully contains even better points) or explore some totally different regions of the search space (as the best-so-far solutions might only be local optima). An EA must be set up in such a way that it solves this dilemma without a priori knowledge of the kind of landscape it will have to explore. The exploitation phase can sometimes be "delegated" to some local optimization procedure, whether called as a mutation operator, or systematically applied to all newborn individuals, moving them to the nearest local optimum. In the latter case, the resulting hybrid algorithm is called a memetic algorithm. In general, there are two driving forces behind an EA: selection and variation. The first one represents a push toward quality and is reducing the genetic diversity of the population. The second one, implemented by recombination and mutation operators, represents a push toward novelty and is increasing genetic diversity. To have an EA work properly, an appropriate balance between these two forces has to be maintained. At the moment, however, there is not much theory supporting practical EA design.

Summary

- Evolutionary algorithms are stochastic search methods that mimic the metaphor of natural biological evolution.
- Evolutionary algorithms include the domains of genetic algorithms (GA), evolution strategies, evolutionary programming, and genetic programming.
- Evolutionary algorithms have a number of components, procedures or operators that must be specified in order to define a particular EA. The most important components of EA are Representation, Evaluation function, Population, Parent selection mechanism, Variation operators, recombination and mutation, Survivor selection mechanism (Replacement) along with initialization and termination conditions.

Review Questions

1. List the crucial components of Evolutionary algorithms.
2. What are the common genetic representations?
3. What are the different fitness assignment techniques?
4. Discuss the issues involved in Population Initialization.
5. What is the significance of rank-based fitness assignment?
6. Is a large sized population always necessary for good solutions? Justify.
7. What is Pareto-ranking?
8. What is stochastic sampling selection sampling method?
9. Discuss the effect of truncation selection.
10. Explain the common selection methods.
11. How does mutation occur?
12. Explain the operation of a reproduction operator.
13. List and explain the different mutation techniques.
14. Illustrate the various crossover types with suitable examples.
15. What is the need for parallel Evolutionary algorithm?
16. How is evolutionary algorithm combined with other optimization techniques?
17. Why is Evolutionary algorithm preferred over other global optimization techniques?
18. What are the critical issues in designing an Evolutionary algorithm?
19. What are the limitations of Evolutionary algorithms?
20. Explain the parallelization methods of Evolutionary algorithms.

Chapter 3
Genetic Algorithms with Matlab

Learning Objectives: On completion of this chapter the reader will have knowledge on:

- Basic definition of Genetic Algorithms
- History of Genetic Algorithm
- Operational functionality of genetic algorithms
- Genetic representation and parameters
- Basics of Schema Theorem and its Mathematical model
- Advanced operators of GA
- Important issues in the implementation of a GA
- Comparison of GA with other methods such as Neural nets, Random search, Gradient methods, Iterated search, Simulated annealing
- Types of genetic algorithm such as Sequential GA, Parallel GA, Hybrid GA, Adaptive GA, Integrated adaptive GA (IAGA), Messy GA, Generational GA (GGA), Steady state GA (SSGA)
- Advantages of GA
- MATLAB illustrative examples of genetic algorithms

3.1 Introduction

Genetic Algorithms (GAs) are a family of computational models inspired by evolution. These algorithms encode a potential solution to a specific problem on a simple chromosome like data structure and apply recombination operators to these structures so as to preserve critical information. Genetic algorithms are often viewed as function optimizers although the range of problems to which genetic algorithms have been applied is quite broad. This chapter deals with the evolution of GA along with the operational functionalities of basic GA. The schema theorem is described with suitable illustrations. Some of the advanced operators that are used in GA are discussed and the types of GA such as parallel, sequential, hybrid, adaptive etc., are

discussed in this chapter. The chapter also provides a set of basic MATLAB illustrations along with suitable codes. An implementation of a genetic algorithm begins with a population of (typically random) chromosomes. One then evaluates these structures and allocates reproductive opportunities in such a way that those chromosomes which represent a better solution to the target problem are given more chances to reproduce than those chromosomes which are poorer solutions.

The goodness of a solution is typically defined with respect to the current population. This particular description of a genetic algorithm is intentionally abstract because in some sense, the term genetic algorithm has two meanings. In a strict interpretation, the genetic algorithm refers to a model introduced and investigated by John Holland (1975). It is still the case that most of the existing theory for genetic algorithms applies either solely or primarily to the model introduced by Holland, as well as variations of genetic algorithm.

In a broader usage of the term, a genetic algorithm is any population based model that uses selection and recombination operators to generate new sample points in a search space. Many genetic algorithm models have been introduced by researchers largely working from an experimental perspective. Many of these researchers are application oriented and are typically interested in genetic algorithms as optimization tools.

Genetic algorithms are search and optimization tools that enable the fittest candidate among strings to survive and reproduce based on random information search and exchange imitating the natural biological selection. The production of new strings in the following generations depend on the initial "parent" strings, the offspring "child strings" are created using parts and portions of the parent strings; the fittest candidate, preserving the best biological features and thus improving the search process. The whole search process is not completely a random process; genetic algorithms utilize the chronological information about old strings in order to be able to produce new and enhanced ones. Genetic algorithms are important because they maintain robustness through evolution. This is at the core of the search process and demonstrates the performance of the algorithm under various circumstances. Robustness in genetic algorithms ensures that simulated models are diverse. Designers can use this robustness to decrease high costs when artificial systems function efficiently, thus creating high level adaptive processes. Computer system designers, engineering system designers, and software and hardware designers will see advancement in the performance, usefulness and robustness of programs through use of the natural model. Significant properties of the natural system such as natural healing, control, management, leadership and reproduction are all benefits that the artificial model borrows from its natural parent model. Studied theoretically and tested experimentally, genetic algorithms are justified not only because they mirror adaptation and survival but also because they prove themselves robust and efficient in complex systems. Through their effective and efficient role in solving search and optimization problems, genetic algorithms are becoming more popular among scientists and engineers and as such, the number of genetic algorithm applications is increasing dramatically. Genetic algorithms require simple computations and yet offer improved and powerful search methodology. Moreover, they do not impose

restrictions on the domain of the search or optimization problem; restrictions found in calculus-based approaches such as continuity, smoothness of the function and the existence of multi-derivatives.

Traditional search and optimization methods can be categorized into two main types: calculus-based and enumerative. Search and optimization methodologies are not necessarily compared at this point. Rather, it is more enlightening to examine the lack of robustness using these techniques in comparison to the effectiveness of genetic algorithms.

Calculus-based Schemes

Perhaps the most popular optimization schemes are the calculus-based; they can be classified as direct search methods and indirect search methods. The latter optimizes an objective function subject to certain constraints (linear and/or nonlinear equations). The Lagrange methodology, which sets the gradient equal to zero, is a methodology commonly used. This approach is a generalization of the idea of searching for local extremal points taught in elementary calculus. The procedure of finding local maxima or minima operates on a smooth and unconstrained function and starts by searching for those points that have derivatives equal to zero. Direct optimizing techniques depend on the value of the local gradient in order to direct searching to find local extremum. Although these techniques were modified, altered and enhanced to become more powerful, there are still some robustness concerns. For one reason, this technique works best on the local scale; the search for a maximum or minimum is performed within the neighborhood of the present point. Optimizing the function around the local maximum point, i.e. searching for the point having a slope of zero, may overlook the absolute maximum point. Moreover, once the local maximum is attained, it may be harder to locate the absolute maximum unless one randomly initializes the procedure again or through a different approach. Ideal optimizing and search problems containing excellent and easily modeled objective functions with ideal constraints and continuous derivatives have attracted mathematicians in the past. However, real-world application problems are full of discontinuities, peaks, huge multi-modal, and chaotic search domains making them difficult to model. Thus, conditions on continuity and the existence of derivatives necessary for optimization and search methodologies limit the use of those techniques to a certain domain of problems. This brings us to the conclusion that calculus-based schemes lack the power to provide robust search through optimization techniques.

Enumerative-based Optimization Schemes

Enumerative techniques are less popular than calculus-based techniques. They depend on either searching a finite continuous highly focused region, or an infinite disconnected search region whereby the optimization algorithm evaluates every point

in the region consecutively. Although the enumerative algorithm in and of itself is simple in nature and sounds like a reasonable process, it is inefficient and thus cannot compete for the robustness race. In reality, enumerative domains of search are considerably enormous or too absolute in focus to perform the search process successfully. Therefore, enumerative-based algorithms are not desirable.

In this section, a detailed description of genetic algorithm, its operators and parameters are discussed. Further, the schema theorem and technical background is also discussed. The different types of GA are also elaborated in detail. Finally MATLAB codes are given for applications such as maximization, traveling sales man problem etc.,

3.2 History of Genetic Algorithm

In the middle of the twentieth century some computer scientists worked on evolutionary systems with the notion that this will yield to an optimization mechanism for an array of engineering queries. GAs were invented and developed by John Holland, his students and his colleagues at the University of Michigan. His team's original intentions were not to create algorithms, but instead to determine exactly how adaptation occurs in nature and then develop ways that natural adaptation might become a part of computer systems. Holland's book *Adaptation in Natural and Artificial Systems* (1975) set forth the lexicon from which all further dialogue concerning GAs would be developed. In essence, his theoretical framework provided the point of reference for all work on genetic algorithms up until recently whereupon it has taken on a new direction, given new technology.

GA, he stated, moves one population of bits (chromosomes and genes) to a new population using a type of "natural selection" along with genetic operators of crossover, mutation and inversion (all biological functions). These operators determine which chromosomes are the fittest and thus able to move on. Although some less fit chromosomes do move forward, on average the most fit chromosomes produce more offspring than their less fit counterparts. Biological recombination occurs between these chromosomes, and chromosomal inversion further completes the process of providing as many types as possible of recombination or "crossover". This remarkable quality that genetic algorithms have of focusing their attention on the "fittest" parts of a solution set in a population is directly related to their ability to combine strings, which contain partial solutions. First, each string in the population is ranked to determine the execution of the tactic that it predetermines. Second, the higher ranking strings mate in couplets.

When the two strings assemble, a random point along the strings is selected and the portions adjacent to that point are swapped to produce two offspring: one which contains the encoding of the first string up to the crossover point, those of the second beyond it, and the other containing the opposite cross. Biological chromosomes perform the function of crossover when zygotes and gametes meet, and so the process of crossover in genetic algorithms is designed to mimic its biological nominative.

Successive offspring do not replace the parent strings; rather, they replace low-fitness ones, which are discarded information at each generation in order that the population size is maintained. Although the mechanics of reproduction through recombination or crossover are relatively simple to understand, they are underestimated in terms of their power for computer applications. After all, it is chance that controls biological evolution and plenty of evidence is available indicating that it is perfectly powerful, even in its simplicity. The mathematician, J. Hadamard, recognized that "there is an intervention of chance but a necessary work of unconsciousness … indeed it is obvious that invention or discovery, be it in mathematics or anywhere else, takes place by combining ideas".

Combinations of ideas, as has been discussed, occur additionally through some mutation of pieces (determined through chance) of information (genes) thus changing the values of some segments (alleles) in the bit strings (chromosomes). Mutation is often misunderstood in the computer/biological lexicon, partly because it is often seen as an event that is abnormal or destructive; visions of giant, radioactive tomatoes come to mind. However, mutation is needed because, even though reproduction and crossover are effective in searching out fit ideas and recombining randomly, unchecked, those systems might overpopulate population samples. Thus, mutations and mutation rates of data are also accounted for in the analysis of population data. Offspring of the mutations in genetic operations are carried on in equations. This recombination of differing parent hypotheses ensures against the evolutionary problem of overcrowding, where a very fit member of the population manages to succeed all other members of the population in creating progeny. This is important because in biological applications as in computational applications, when the diversity of the population is reduced, operational progress is slowed.

3.3 Genetic Algorithm Definition

The *genetic algorithm* is top-most entity in any Evolutionary implementation. There can be only one genetic algorithm construct in any Darwin program and this construct should contain at least one population as a member. It is responsible for creating its population and evolving it until a termination condition is reached. The genetic algorithm also specifies the replacement strategy. The genetic algorithm has *terminator, replacement, evolver, initializer* and *printer* as its moderators. The syntax of the *genetic algorithm* is as follows:

alg_typedef:	*algorithm id*
	algorithm moderators
algorithm moderators:	*initializer prototype*
	printer prototype
	evolver prototype
	terminator prototype
	replacement prototype

The genetic algorithm initializer is responsible for initializing its population and parameters. The default is to call the population initializer to initializer its population.
The genetic algorithm terminator is a boolean function deciding when a termination condition is reached. The default stops if number of generations is 100.
The genetic algorithm replacement moderator is used to decide if which of the chromosomes in the parent and offspring populations should survive. The generated replacement moderator is dummy, so the offspring replaces its parent population.
The genetic algorithm evolver moderator is the heart of any Darwin program. This moderator is responsible for first creating and then evolving a population until the termination moderator decides stopping this process. The generated genetic algorithm evolver implements Simple GA.
The genetic algorithm printer provides formatted output of the genetic algorithm construct. By default, all the algorithm members, including the population, are printed. This moderator is called when the termination condition of the genetic algorithm is reached.

3.4 Models of Evolution

In traditional GAs the basic evolution step is a generation of individuals. That means that the atomic advance step of the GA is one generation. There exists another interesting alternative called Steady-State GAs in which the atomic step is the computation of one single new individual. In these algorithms after generating the initial population, the GA proceeds in steps that consist of selecting two parents, crossing them to get one (or two) single individual and mutating the resulting offspring. The new string is inserted back in the population and one of the pre-existing strings (a random string or the worst present string) leave the pool (normally if the new string is worst than the actual worst it is not inserted at all).

This one-at-a-time reproduction was initially introduced to overcome problems in the training of neural networks and was extended from there to the rest of present applications. It represents an interesting equilibrium between the exploration and exploitation edges of a GA, because the GA maintains a pool of strings (exploration, schemata, etc…) and at the same time it performs some kind of hill-climbing since the population is only pushed toward better solutions. Elitism and ranking are very common to steady-state GAs.

Finally there exists an intermediate model of evolution for a sequential GA in which a percentage of the full population is selected for the application of genetic operations. This percentage is called the Genetic Algorithm Percentage (GAP) and it represents the generalization of any evolution model since it includes the two opposite approaches (generations versus one-at-a-time) and any other intermediate model.

3.5 Operational Functionality of Genetic Algorithms

A genetic algorithm is an iterative procedure that consists of a constant-size population of individuals, each one represented by a finite string of symbols, known as the *genome*, encoding a possible solution in a given problem space. This space, referred to as the *search space*, comprises all possible solutions to the problem at hand. Generally speaking, the genetic algorithm is applied to spaces which are too large to be exhaustively searched. The symbol alphabet used is often binary, though other representations have also been used, including character-based encodings, real-valued encodings, and most notably tree representations.

The standard genetic algorithm proceeds as follows: an initial population of individuals is generated at random or heuristically. Every evolutionary step, known as a *generation*, the individuals in the current population are *decoded* and *evaluated* according to some predefined quality criterion, referred to as the *fitness*, or *fitness function*. To form a new population (the next generation), individuals are *selected* according to their fitness. Many selection procedures are currently in use, one of the simplest being Holland's original *fitness-proportionate selection*, where individuals are selected with a probability proportional to their relative fitness. This ensures that the expected number of times an individual is chosen is approximately proportional to its relative performance in the population. Thus, high-fitness ("good") individuals stand a better chance of "reproducing", while low-fitness ones are more likely to disappear.

Selection alone cannot introduce any new individuals into the population, i.e., it cannot find new points in the search space. These are generated by genetically-inspired operators, of which the most well known are *crossover* and *mutation*. Crossover is performed with probability p_{cross} (the "crossover probability" or "crossover rate") between two selected individuals, called *parents*, by exchanging parts of their genomes (i.e., encodings) to form two new individuals, called *offspring*; in its simplest form, substrings are exchanged after a randomly selected crossover point. This operator tends to enable the evolutionary process to move toward "promising" regions of the search space. The mutation operator is introduced to prevent premature convergence to local optima by randomly sampling new points in the search space. It is carried out by flipping bits at random, with some (small) probability p_{mut}. Genetic algorithms are stochastic iterative processes that are not guaranteed to converge; the termination condition may be specified as some fixed, maximal number of generations or as the attainment of an acceptable fitness level. The standard genetic algorithm in pseudo-code format is shown below:

```
begin GA
 g:=0 { generation counter }
 Initialize population P(g)
 Evaluate population P(g) { i.e., compute
   fitness values }
 while not done do
   g:=g+1
   Select P(g) from P(g−1)
```

```
    Crossover P(g)
    Mutate P(g)
    Evaluate P(g)
   end while
end GA
```

3.6 Genetic Algorithms – An Example

Consider a cylindrical closed container (Figure 3.1), which can be determined by the two variables, height h and radius r. For simplicity, it is assumed that the other parameters exist, but are nuisance factors including the thickness, shape and material characteristics. Without loss of generality, suppose the container has volume V of at least 0.1 liters. The purpose is to minimize the material cost C of this container. So, this information is translated into an optimization model consisting of objective and constraint functions called the non-linear programming problem.

The problem is defined as:

Minimize: $C(r,\ h) = \gamma\,(2\pi rh + 2\pi r2) = 2\pi r\gamma\,(h+r)$,
subject to: $V\ (r,h) = r^2h\pi \geq 0.1$ and
bounding variables: $r_{\min} \leq r \leq r_{\max},\ h_{\min} \leq h \leq h_{\max}$,

where γ is the material cost per cm^2, r, h are the height and radius, respectively, in cm^2. Optimize cost C, a function of h and r satisfying the constraint V. Suppose the cost of the container be 5 units. To proceed with the genetic algorithm, a transformation is done from decimal system to binary system.

Consider two 5-bit binary strings corresponding to the two variables h and r with a total string length of 10. Also, assign the container a height of 7 cm and radius of 3 cm.

$$\underbrace{00011}_{r}\,\underbrace{00111}_{k}$$

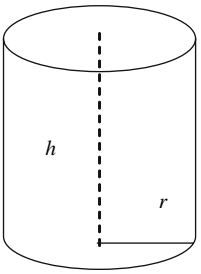


Fig. 3.1 Cylindrical container with radius r and height h

The height and radius are given by $(r,h) = (3,7)$ cm. The binary chromosomal "string" representation is given by 00011 00111. In the 5-bit representation, both the h and r have a lower limit of 0 and an upper limit of 31, which implies that there are $2^5 = 32$ distinct possible solutions. This gives the genetic algorithm the integer interval $[2^0 - 1,\ 2^5 - 1] = [0,31]$ from which to select. On the other hand, the genetic algorithm is; in general, not limited to this set of integers and requires larger sets that contain both integer and non-integer quantities. This can be achieved simply by considering strings of longer lengths providing wider lower and upper limits. The following is a transformation to obtain the value of any parameter:

$$t_n = t_n^{min} + \frac{t_n^{\max} - t_n^{\min}}{2^{kn} - 1}.\varphi\,(\ln)$$

where t_n is the nth parameter value, $t_n^{\min} = 0$ is the minimum value the nth parameter can hold, $t_n^{\max} = 31$ is the maximum value the nth parameter can hold, $kn = 5$ is the length of the nth string, *.(ln)* is the decoding value of the string *ln*. This transformation has the subsequent properties:

(i) Variables can hold positive values or negative values.
(ii) A well-defined string length can be used to obtain a variable with randomly finite accuracy.
(iii) Various string lengths can be used to obtain different accuracies for unlike variables.

The coding stage of the variables allows us to obtain a pseudo-chromosomal description of the model in a binary string. In this problem that is being modeled, one may biologically represent the container with radius $r = 3\,\text{cm}$ and height $h = 7\,\text{cm}$ using the 10-bit chromosome "string". In the case of the can, the container is the phenotypic representation consisting of 10 genes, which build the simulated chromosome. The leftmost bit "gene" of the chromosome of radius r allows one to observe how the shape "phenotype" of the container is determined by the 10 genes. If the value of this bit, perhaps the most important bit "gene", is zero then the radius can hold values that range between 0 cm and 15 cm. If the value is one, then the radius can hold values that range between 16 cm and 31 cm. As a result, this gene "bit" controls the narrowness of the container. Thus, the zero value means the container is narrow and one value means the container is wide. Any permutation of the other genes may result in the different characteristics of the container, some of which are desirable properties and others might not be that useful. Hence, one has transformed the produced children into a string representation. Furthermore, one proceeds in the same process applying genetic operations to produce improved and desirable offspring, which requires us to define and assign fitness values of goodness; in the form of a string, accompanied with each child, offspring. Once the genetic algorithm finds a solution to the optimization problem, it is essential to rerun the algorithm using this solution as the new values of the variables. It is recommended to evaluate the objective function and the constraint using the produced string "solution". If the optimization problem has no constraints, then one assigns the fitness of the string a function value that corresponds to the solution of the objective function. In general,

this value equals to the value of the objective function. As an illustration, consider the above example where the container is characterized by the 10-bit string, which has a fitness value of $f(l) = 2 \cdot \pi \cdot 3 \cdot 0.0254 \cdot (7+3) = 5$. Again, the goal is to minimize cost of the objective function. Therefore, when the fitness value is small, a better solution is guaranteed.

3.7 Genetic Representation

Genetic representation is a way of representing solutions/individuals in evolutionary computation methods. Genetic representation can encode appearance, behavior, physical qualities of individuals. Designing a good genetic representation that is expressive and evolvable is a hard problem in evolutionary computation. Difference in genetic representations in one of the major criteria drawing a line between known classes of evolutionary computation.

Genetic algorithm uses linear binary representations. The most standard one is an array of bits. Arrays of other types and structures can be used in essentially the same way. The main property that makes these genetic representations convenient is that their parts are easily aligned due to their fixed size. This facilitates simple crossover operation. Variable length representations were also explored in Genetic algorithms, but crossover implementation is more complex in this case.

Human-based genetic algorithm (HBGA) offers a way to avoid solving hard representation problems by outsourcing all genetic operators to outside agents (in this case, humans). This way the algorithm need not be aware of a particular genetic representation used for any solution. Some of the common genetic representations are

- binary array
- genetic tree
- parse tree
- binary tree
- natural language

3.8 Genetic Algorithm Parameters

Many researchers, including DeJong (1975), have worked on the parameters of the operators of mutation and one-point crossover and found out that the following values provide desirable results. DeJong set the population size to be 50, the crossover probability *P*x to be 0.6, the mutation probability *P*m to be 0.001, elitism to be 2 and no windowing.

3.8.1 Multi-Parameters

To obtain parameters with optimal results, one may opt for an integer from the interval [0, 15], which is equivalent to $[0,\ 2k-1]$ where k is the length of the string. In this case, a parameter of 4 bits is used for encoding. However, a new approach can

be used to encode optimization problems with real multi-parameters. This technique is called the concatenated, multi-parameter, mapped, fixed-point coding. Instead of working on $[0,\ 2^k - 1]$, the method maps $[0,\ 2k]$ to the interval $[I_{\min},\ I_{\max}]$ using a linear transformation. This allows us to monitor the decoded parameter including its precision and range; the following formula can be used to compute the precision R:

$$R = \frac{I_{\max} - I_{\min}}{2^k - 1}$$

The procedure to build the multi-parameter code is straightforward. Consider a single parameter code, to proceed with the process take one of these code blocks which has a length, $I_{\min}$ and $I_{\max}$ then start concatenating them until reaching the desired point.

3.8.2 Concatenated, Multi-Parameter, Mapped, Fixed-Point Coding

Consider a single-parameter code I_n of length $k_n = 3$. This parameter has corresponding $I_{\min} = 000$ and $I_{\max} = 111$. Then a linear transformation of a multi-parameter code consisting of 9 parameters would be

$$001001 \cdots 100111$$
$$I_1 I_2 \cdots I_8 I_9.$$

3.8.3 Exploitable Techniques

Although, inversion, addition and deletion are exploitable techniques their impact is still uncertain. Inversion is basically taking a cross section of a given string then rearranging the characters of this section in a reverse order. The concept of inversion guided researchers to both the uniform ordered-base and the partially mixed crossovers, which discussed earlier.

Inversion Operator

Inversion operator

$\nu \quad \eta\|\alpha\mu\gamma\kappa\iota\|o\xi\zeta$	$\nu \quad \eta\|\iota\kappa\gamma\mu\alpha\|o\xi\zeta$
Before inversion	After inversion

Addition operator

$\nu \quad \eta\alpha\mu\gamma\kappa\iota o\xi\zeta$	$\nu \quad \eta\alpha\mu\gamma\ \mathrm{M}\kappa\iota o\alpha\xi\zeta$
Before addition	After addition

Deletion operator

ν ηαμγκιοξζ	ν αμγκιοξ
Before deletion	After deletion

One performs the function of deletion by randomly selecting and deleting one or more characters of the targeted cross section of a given string, however, addition can be done by randomly making copy of a certain permitted character and then inserting this character randomly into a targeted position. It is note worthy to mention that the addition and deletion operators have direct impact on the length of the string, which calls for adapting the techniques including the fitness.

3.9 Schema Theorem and Theoretical Background

A schema (singular), as understood in schema theory, represents generic knowledge. A general category (schema) will include slots for all the components, or features, included in it. Schemata (plural) are embedded one within another at different levels of abstraction. Relationships among them are conceived to be like webs (rather than hierarchical); thus each one is interconnected with many others.

A simple way to view the Genetic Algorithm is provided by schema theorem. It explains how crossover allows a genetic algorithm to zero in on an optimal solution. However, schema is inadequate in determining some characteristics of the population. The Schema Theorem is the fundamental theorem in genetic algorithms. John Holland was the first to initiate the Schema Theorem in 1975. He first introduced the concept of "implicit parallelism", which explains the process of genetic algorithm. Implicit parallelism is the fundamental foundation for genetic algorithms. The ability to determine all the possible solutions concurrently is perhaps the most significant quality genetic algorithms can provide. Strings, in general, contain an array of building blocks, which are evaluated immediately using implicit parallelism, however, genetic algorithm measures these blocks simultaneously when determining string's fitness. The process of transferring the building blocks to the following generation pioneers when the algorithm starts searching for patterns and similarities inside the string. Unlike traditional algorithms, this procedure enhances the performance of genetic algorithms significantly.

Consider a binary string with $2k$ different degrees of freedom; subsequently this string includes $2k$ possible schemata. Hence, one is able to obtain promising information about the schema and its fitness. Genetic algorithms can be thought of as a competition among schemata with $2k$ distinct competitions. The Schema Theorem explains the increase of a schema from a particular generation to the subsequent one. The Schema Theorem is frequently explained such that concise, low-order, short schemas whose mean fitness stays beyond the average will be given exponential increasing number(s) of samples beyond the instance of time. The quantity of the assessed occurrence of the schemas that are not disrupted and which stay above the mean in fitness is enhanced by a factor of $f(S)/f_{\text{pop}}$ for every generation, where

$f(S)$ is the fitness mean value of schema S, and f_{pop} is the fitness mean value of the population.

The Schema Theorem deals with only the damaging effects of recombination and mutation and is a lower bound. On the other hand, recombination is the chief source of the genetic algorithm's strength. It has the ability to put together situations with decent schemas, which to develop various occurrences of equally decent or more enhanced higher-order schemas. The procedure by which genetic algorithms operate is known as the Building Block Hypothesis. The low-ordered and above-average schemata grow exponentially in subsequent generations are referred to as *building blocks*.

3.9.1 Building Block Hypothesis

The credibility of the GA does not rest solely on the schema theorem. It also rests on the so-called building-block hypothesis. This states that the crossover GA works well when short, low-order, highly fit schemas recombine to form even more highly fit, higher-order schemas. In fact, as Forrest and Mitchell (1996) note, "the ability to produce fitter and fitter partial solutions by combining blocks is believed to be the primary source of the GA's search power". Unfortunately, while examining the assumptions introduced by the building-block hypothesis, it is found that they contradict those introduced by the schema theorem.

The building-block hypothesis assumes that the other blocks on the genotype typically affect the fitness of any one block. If this were not the case it would be meaningless to talk about a "building-block process" operating over and above the usual evolutionary process. Thus the building-block hypothesis implicitly assumes only a positive effect of epistasis on fitness and thus contradicts the low-epistasis assumption introduced by the schema theorem. While considering the length implications of the building-block hypothesis a further contradiction is uncovered. During the building-block process, the schemas that require processing at any given stage are actually the blocks that have been put together by the prior building-block process. Except at the initial stage, the defining length of these schemas is related to the defining lengths of the *components* of the blocks. Consider a block made up of just two schemas. One schema may be nested inside the other. In this case the defining length of the block is simply the defining length of the longer of the two schemas. At the other extreme the two schemas might be situated at opposite ends of the genotype. In this case the defining length of the new block will be close to the genotype length.

If a conservative assumption is made such that, on average, the defining lengths of blocks will be at least the *sum* of the defining lengths of the components, then it is found that real schema length will increase at least exponentially during the building-block process. Assume all original schemas have defining length 1 and that all blocks have two components. Then a first-level block has defining length 2. A second level block has defining length 4. A third level block has defining length 8, and so on.

The schemas that must be processed at any given stage are the blocks that have been created by previous processing. Thus, strictly speaking, the schema growth formula - which defines growth in terms of a *particular* schema - cannot be meaningfully applied to a GA in a building-block scenario. If it is to be applied, then a time-fixed schema identifier and a definition of defining length which explicitly captures the time-related growth should be used. For example, let H_t denote a schema processed at time t, and define the schema length as:

$$\delta(H_t) = x^t$$

with x being the defining length of an original schema.

Once, the implicit growth in schema lengths is made explicit, it is found that that the building-block hypothesis (which depends on negligible schema length) is very likely to be violated in all but the initial stage of processing. It demands increasing schema length and thus violates the shortness assumption introduced by the schema theorem.

Thus both of the assumptions introduced by building-block hypothesis directly contradict assumptions introduced by the schema theorem. The situation is illustrated schematically in Figure 3.2.

Given the importance of the building-block hypothesis within the GA paradigm this clash of assumptions occurring at the most fundamental level of the analysis is of particular interest. As Forrest and Mitchell (1996) have commented there is a "need for a deeper theory of how low-order building blocks are discovered and combined into higher-order schemas" which is beyond the scope of this book.

3.9.2 Working of Genetic Algorithms

The pioneer of genetic algorithms (GA), John Holland, was the first person to fully describe GAs mathematically. GAs are designed to search for, discover, emphasize and breed good solutions (or "building blocks") to a problem in a highly parallel

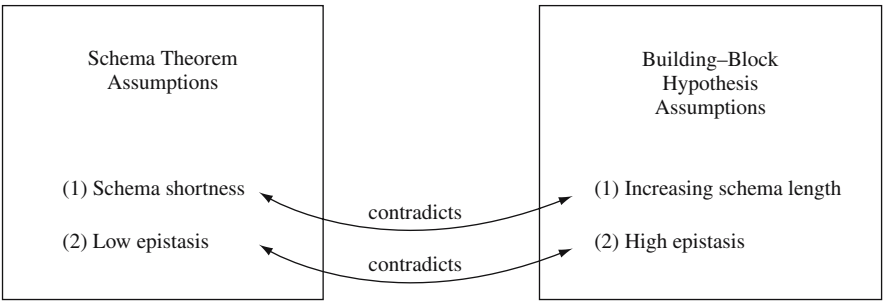


Fig. 3.2 Inherent contradictions in the schema/building-block framework

fashion. Holland invented the idea of *schemas* (or *schemata*) to formally conceptualize the notion of "building blocks." Each schema is a template for a bit string of length *l*. The schemata are comprised of one's and zero's (defining the bits), and asteriks ('*') that act as wild cards within the string. Therefore, the schema:

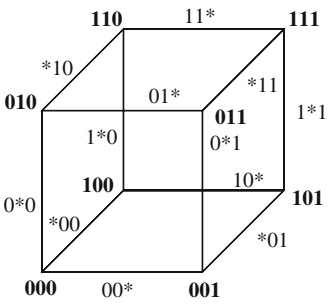
$$H = 1^{***}0^{*}$$

represents all bit strings that begin with one and have a '0' at the 5th bit position. Examples of bit strings that satisfy this are 111001, 100001, 111100, 111101 etc. These are called *instances* of the schema. The H in the equation stands for *hyperplane*. A hyperplane is a plane of various dimensions - the dimension of the plane is equal to *l*, the number of bits in the string. To define this principle further, take the schema: H $=^{**}$ 0. Since H is a three-dimensional hyperplane it can be graphed as shown in Figure 3.3:

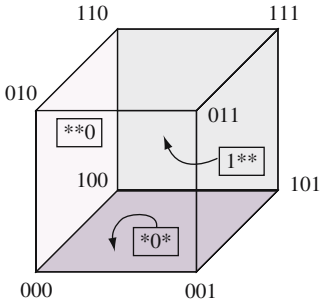
Other traits of a schema include the *defined bits* and the *defining length*. The defined bits are the number of bits defined within the schema. The defining length is the distance between the two most outer defined bits.

3.9.3 Sets and Subsets

Most of the subsets of a length *l* bit string cannot be described as a schema. For a bit string of length-*l*, there are 2^{l} possibilities, $2^{2\wedge l}$ possible substrings (because of the



Search space for a problem with L = 3 and K = 2 (binary alphabet)



Three of the six schemata of order 1

Order of the Schema o(H)	Hyperplane Dimension	Geometric Interpretation	Strings in the Schema	Number of Schemata
3	0	Point	1	8
2	1	Line	2	12
1	2	Plane	4	6
0	3	Cube	8	1
			Total	27

Fig. 3.3 The concept of schemata

wildcards). Therefore, for a bit string with 4-bits, there are 16 possible bit strings, an amazing 65536 substrings, yet only 81 schemata to represent them.

Any bitstring is a member of 2^l different schemata. The length-2 bitstring '11' is an instance of **, 1^*, ^{*1}, and 11. Now, in a population of n-strings it is easy to see that there can be anything between 2^l and $n \times 2^l$ different schemata. It can therefore be said, that while the GA explicitly evaluates n strings during any given generation, it is implicitly evaluating a much larger number of schema. Think of it as half of the population being part of the schema '$1^{**}\ldots^*$' and the other half as '$0^{**}\ldots^*$'. Now, after a given generation has been run, the average fitness of those schemata has to be evaluated (it will be quite an erroneous estimate, since the population actually evaluated was only a tiny portion of the actual schema). The estimates of these averages of obviously not stored explicitly since the schemata themselves are not explicitly evalulated. Where this idea is of help, is in looking at the increase or decrease of a given schema in a population - because it can be described as though the GA were storing these implicitly calculated values.

3.9.4 The Dynamics of a Schema

The method of approximating the dynamics can be calculated through the following method. Let H be a schema with at least one instance in the given population at the time t. Let the function m (H,t) be the number of instances of H and time t, and let the function û (H,t) be the observed average of H and time t (this is the phenomena described in the above section). Now, it is required to calculate the expected number of instances of schema H during the next time interval. Assuming that E(x) is the expected value of x, E $(m(H,t+1))$ has to be determined.

Now, the expect value of number of instances at $t+1$ is equivalent to the number of offspring produced. Therefore, in order to calculate further the form of selection process that is to be used has to be determined. If a *fitness-proportionate* selection method is to be used, then the expected number of offspring is equal to the fitness of the string over the average fitness of the generation or:

$$fitness - proportionate = \frac{f(x)}{\hat{f}(t)}$$

So, assuming that xϵH (x is a subset, or instance, of H), the equation obtained is:

$$E(m(H,t+1)) = \sum_{x \in H} \frac{f(x)}{\hat{f}(t)}$$

Now, using the definition of û (H,t) the final equation is:

$$\sum_{x \in H} \frac{f(x)}{m(H,t)} = \hat{u}(H,t)$$
$$\therefore E(m(H,t+1)) = \frac{\hat{u}(H,t)}{\hat{f}(t)} m(H,t)$$

Therefore the basic formula for tracking the dynamics of a schema in a genetic algorithm is obtained. Genetic algorithms are simple, since they also simulate the effects of mutation and crossovers. These two effects have both constructive and destructive effects – here the destructive effects are considered.

3.9.5 Compensating for Destructive Effects

Considering crossover, let p_c be the probability that crossover will be applied to a given string in the population, and assume that an instance of schema H is picked to be the parent. Schema H is said to have "survived" if one of the offspring is still an instance of H. For example, given the schema "11^{*****}", and given two strings, 110010 and 010100.It is found that the first bit is an instance of the schema, the second is not. Now, if a single-point crossover is applied at the 4^{th} bit, the following offspring is obtained - 110000 and 010110. The first child is still an instance of the schema, thus it has survived.

Given $S_c(H)$ as being the probability of schema H surviving a single-point crossover, d(H) as the defining length, and l as the length of the bit string:

$$S_C(H) \geq 1 - p_c * \frac{d(H)}{l-1}$$

A breakdown of this equation is necessary. The d(H)/$(l-1)$ is equal to the fraction of the string that the schema occupies. For example, if a defining length of 3 is present in a string of length 5, then 75% of the string is part of schema H. Now, the result is multiplied by the probability that they survive, p_c. This gives us an upper-bound that the schema will be destroyed. To obtain the lower-bound, the upper bound is subtracted from 1 (cumulative probabilities is equal to 1).

Considering mutation, let p_m be the probability that a bit will be mutated, and that the function $S_m(H)$ is the probability that the schema H will survive the mutation. In order to survive the mutation, the schema must remain intact, therefore $S_m(H)$ must be equal to the probability that a bit will NOT be mutated $(1-p_m)$ raised to the power of the number of defined bits in the schema, or:

$$S_m(H) = (1-p_m)^{o(H)}$$

where o(H) is the number of defined bits (or *order*) of the schema. From these two calculations it is found that the shorter and lower order the schema, the greater chance it has to survive. By applying the effects with the above equations, the Schema Theorem derived by Holland is obtained as:

$$E(m(H,t+1)) \geq \frac{\hat{u}(H,t)}{\hat{f}(t)} m(H,t)(1-p_c\frac{d(H)}{l-1})\left((1-p_m)^{o(H)}\right)$$

Crossover is regarded to be one of the chief areas of the GA's power, and mutation is often seen as an insurance policy to prevent a loss of diversity. It is the characteristic of the GA to be able to *implicitly evalulate* many solutions that makes it so powerful.

The short, low order definition schemata with average fitness above the mean population fitness get a number of instances that grow exponentially. The GA makes this work in an implicit parallel way for all the population and schemata. Figure 3.4 summarizes the schema theorem and its effects.

3.9.6 Mathematical Models

The Schema Theorem only shows how schemas dynamically change, and how short, low-order schemas whose fitness remain above the average mean receive exponentially growing increases in the number of samples. It cannot make more direct predictions about the population composition, distribution of fitness and other statistics more directly related to the genetic algorithm itself. The *simplest* mathematical model for an infinite population GA as derived by Michael Vose and Gunar Liepins is dealt in this section. The algorithm is as follows:

1. Calculate the fitness of each string.
2. Choose two parents using fitness-proportionate selection.
3. Use a single-point crossover, use only one of the offspring.
4. Mutate each bit with probability p_m, and place in population.
5. Go to step 2 until population complete.
6. Go to step 1.

THE SCHEMA THEOREM

SELECTION EFFECTS

$$fitness - proportionate = \frac{f(x)}{\hat{f}(t)}$$

CROSSOVER EFFECTS

$$S_C(H) \geq 1 - p_c * \frac{d(H)}{l-1}$$

MUTATION EFFECTS

$$S_m(H) = (1-p_m)^{o(H)}$$

COMBINED EFFECTS: THE SCHEMA THEOREM

$$E(m(H,t+1)) \geq \frac{\hat{u}(H,t)}{\hat{f}(t)} m(H,t)(1-p_c \frac{d(H)}{l-1}) \left((1-p_m)^{o(H)}\right)$$

Fig. 3.4 Derivation of the schema theorem

As always, this applies to a random population of binary strings of length l. In the model that Vose and Liepens used, the binary strings simply represented binary numbers. Therefore, when $l = 8$, $00000101 = 5$. The population of strings was represented at any generation t by two vectors, p(t) and s(t). The i^{th} component of p(t) denotes the proportion of the population at t consisting of string i (this is denoted as $p_i(t)$) The i^{th} component of s(t) denoted the probability of the string mating. Both p(t) and s(t) are column vectors even though they are written as row vectors. It is necessary to know this because the fitnesses are represented in a two-dimensional matrix F. F is structured such that $F_{i,j}$ is zero for all elements, except when $i = j$. In this case, the element is the fitness of string i. Therefore, using the definition of fitness-proportionate selection:

$$\vec{s}\,(t) = \frac{F\,\vec{p}\,(t)}{\sum_{j=0}^{2^l-1} F_{jj} p_j(t)}$$

Now that p(t) and s(t) can be defined in terms of each other, an operator G that acts like a genetic algorithm is determined. It implies that when G was applied to s(t), it would mimic the effects of running a genetic algorithm through one generation:

$$\vec{s}\,(t+1) = G\,\vec{s}\,(t)$$

Now, using the idea that the expected value of the population at t + 1 is equal to the probability of the strings mating, let's define x ∼ y as 'x differs from y by a scalar factor. Thus:

$$E(\vec{p}\,(t+1)) = \vec{s}\,(t)$$
$$\therefore \vec{s}\,(t+1) \sim F\,\vec{p}\,(t+1)$$
$$\therefore E(\vec{s}\,(t+1)) \sim F\,\vec{s}\,(t)$$

The way Vose and Liepens continued was by defining G as the composition of the fitness matrix F and a recombination operator (we'll call this β). ß mimic both crossover and mutation, thus the term 'recombination.' Vose and Liepens defined ß by finding $r_{i,j}(k)$, the probability that string k will be produced by recombination between string i and string j. Therefore, once $r_{i,j}(k)$ is known it is found that:

$$E(p_k(t+1)) \sum_{i,j} S_i(t) S_j(t) r_{i,j}(k)$$

This equation shows that the expected value of the proportion of string k in the next generation is equal to the "probability that it will be produced by each given pair of parents, times by those parents' probabilities of being selected, summed over all possible pairs of parents."

Vose and Liepens firstly defined a simplier matrix M whose elements at i,j gave the probability $r_{i,j}(0)$ that string 0 (all bits 0) would be produced through

recombination of i and j. $r_{i,j}(0)$ is equal to the sum of the probability that crossover does not occur between i and j and the selected offspring is mutated to all zeros, and the probability that crossover does occur and the selected offspring is mutated to all zeros. So, the probability that crossover occurs between i and j is p_c (thus the probability it doesn't occur is $1-p_c$). Same goes with mutation, p_m it does occur, $1-p_m$ is doesn't occur. Let's define $|\mathrm{i}|$ as the number of ones in the bit-string i of length l. Therefore, the probability that i will be mutated to all zeros is:

$$p_m^{|i|}(1-p_m)^{t-|i|}$$

that is, the probability that all of the $|i|$ ones are mutated multiplied by the probability that none of that $(1-|i|)$ zeros are mutated. With this in mind, the first term in the expression for $r_{i,j}(0)$ equates to:

$$\frac{1}{2}(1-p_c)\left(p_m^{|i|}(1-p_m)^{l-|i|}+p_m^{|j|}(1-p_m)^{l-|j|}\right)$$

$|j|$ is the same as $|i|$, only defined for string j. It is found that 1/2 is included since only one of the children survive the mating process. Now, for the second term let h and k denote the two offspring produced from a single-point crossover at point c (counted from right-hand side, like binary). There are, therefore, $l-1$ crossover points, each having the probability $1/(l-1)$ of being chosen. Therefore, the second term can be written as:

$$\frac{1}{2}\frac{p_c}{l-1}\sum_{c=1}^{l-1}p_m^{|h|}(1-p_m)^{1-|h|}+p_m^{|k|}(1-p_m)^{l-|k|})$$

To define h and k, let i_1 be the substring of i consisting of $l-c$ bits to the left of c, and i_2 to the substring of i to the right of c. Therefore,$|h|=|i|-|i_2|+|j_2|$, and $|k|=|j|-|j_2|+|i_2|$. Now, using a bit of fancy notation adjustment, you can simplify the equation down a little:

$$\begin{aligned}|i_2| &= |(2^c-1)\wedge i| \qquad \textit{where}\wedge\textit{is the bitwise AND operator.}\\ \therefore\Delta_{i,j,c} &= |i_2|+|j_2| = |(2^c-1)\wedge i|-|(2^c-1)\wedge j|\\ \therefore|h| &= |i|-\Delta_{i,j,c}\ \textit{and}\ |k| = |j|-\Delta_{i,j,c}\end{aligned}$$

Finally to define $r_{i,j}(0)$, lets define η as $p_m/(1-p_m)$. After a little of algebraic manipulation the final equation is:

$$\begin{aligned}r_{i,j}(0) = \frac{(1-p_m)^l}{2}\Bigg[&\eta^{|i|}\left(1-p_c+\frac{p_c}{l-1}\sum_{c=1}^{l-1}\eta^{-\Delta_{i,j,c}}\right)\\ &+\eta^{|j|}\left(1-p_c+\frac{p_c}{l-1}\sum_{c=1}^{l-1}\eta^{-\Delta_{i,j,c}}\right)\Bigg]\end{aligned}$$

Using $r_{i,j}$, an expression for G is also obtained.

$G(\vec{x}) = F \circ \beta(\vec{x})$ *where* $\circ$ *is a composition operator*

$G(\vec{s}\ (t)) \sim \vec{s}\ (t+1)$ *when applied to a limit of an infinite population*

Defining $|v|$ as the sum of its elements, G_p is defined as:

$$G_p(\vec{x}) = \beta\left(\frac{F\ \vec{x}}{|F\ \vec{x}\ |}\right)$$

$$\therefore G_p\left(\vec{p}\ (t)\right) = \vec{p}\ (t+1)$$

Although G and G_p act on different representations of the population, they can be translated into one and other through simple transformation.

3.9.7 Illustrations based on Schema Theorem

Illustration 1:

1. Consider the following two schema

$^{*****}111$ and $1^{*****}1^{*}$

(a) What is the defining length? What is the order?

(b) Let $p_m = 0.001$ (probability of simple mutation). What is the probability of survival of each schema?

(c) Let $p_c = 0.85$ (probability of cross-over). Estimate the probability of survival of each schema?

Solution:

(a).

Order:

$o(H_1) = 3;\ o(H_2) = 2;$

Defining Length:

$\delta(H_1) = 2;\ \delta(H_2) = 6;$

(b).

Given: $p_m = 0.001$

Mutation: $p_s(S) = 1 - o(S)p_m$

$p_s(H_1) = 1 - o(H_1)p_m = 1–3^{*}0.001 = 99.7\%$

$p_s(H_2) = 1 - o(H_2)p_m = 1–2^{*}0.001 = 99.8\%$

(c).

Given: $p_c = 0.85;$

We know that $m = 8;$

$$\text{Cross} - \text{over}: \ p_s\ (S) = 1 - p_c\frac{\delta(S)}{m-1}$$

$$p_s\ (H_1) = 1 - p_c\frac{\delta(H_1)}{m-1} = 1 - 0.85^{*}\frac{2}{8-1} = 76\%$$

$$p_s\ (H_2) = 1 - p_c\frac{\delta(H_2)}{m-1} = 1 - 0.85^{*}\frac{6}{8-1} = 27\%$$

Illustration 2:

A population contains the following strings and fitness values at generation 0:

S No	String	Fitness
1	**10001**	**20**
2	**11100**	**10**
3	**00011**	**5**
4	**01110**	**15**

Calculate the expected number of strings covered by the schema 1^{****} in generation 1 if $p_m = 0.01$ and $p_c = 1.0$.

Solution:

Schema-Theorm:

$$\xi(S,t+1) \geq \xi(S,t).\frac{eval(S,t)}{\overline{F(t)}}\left[1 - p_c.\frac{\delta(S)}{m-1} - o(S).p_m\right]$$

$$eval(H,0) = \frac{20+10}{2} = 15$$

$$\overline{F(t)} = \frac{20+10+5+15}{4} = 12.5$$

The number of strings covered by the given schema in generation 1 is :

$$\xi(H,1) = 2 * \frac{15}{12.5} * \left[1 - 1.0 * \frac{0}{4} - 0.01 * 1\right] = 2 * 12 * (1 - 0 - 0.01) = 2.376$$

Illustration 3:

Suppose a schema H which, when present in a particular string, causes the string to have a fitness 25% greater than the average fitness of the current population. If the destruction probabilities for this schema under mutation and crossover are negligible, and if a single representation of the schema is present at generation 0, determine when the schema H will overtake the populations of size $n = 20; 50; 100; 200$.

Solution:

$$\xi(H,t) \geq \xi(H,t-1) * \frac{\overrightarrow{eval(H,t-1)}}{\overline{F(t-1)}} * 1$$

$$1.25$$

$$\xi(H,t-2) * 1.25 * 1.25$$

$$\vdots$$

$$\xi(H,0) * (1.25)^t$$

$$(1.25)^t$$

Therefore,

$$
\begin{aligned}
& n \leq (1.25)^t \\
& t \geq \log_{1.25}(n) \\
& \geq \frac{\log(n)}{\log(1.25)}
\end{aligned}
$$

The schema will overtake the populations when,

$$
\begin{aligned}
& n = 20 \Rightarrow t \geq 14 \\
& n = 50 \Rightarrow t \geq 18 \\
& n = 100 \Rightarrow t \geq 21 \\
& n = 200 \Rightarrow t \geq 24
\end{aligned}
$$

Illustration 4:

Suppose a schema H which, when present in a particular string, causes the string to have fitness 10% less than the average fitness of the current population. Suppose that the schema theorem is equality instead of an inequality and that the destruction probabilities for this schema under mutation and crossover can be ignored. If representatives of the schema are present in 60% of the population at generation 0, calculate when schema H will disappear from the population of size n = 20; 50; 100; 200.

Solution:

$$
\begin{aligned}
& \xi(H,t) = \xi(H,0) * (0.9)^t \\
& = 0.6 * n * (0.9)^t \quad 0.6 * n * (0.9)^t \\
& \quad \frac{1}{2} \\
& \quad\quad \frac{1}{1.2 * n} \\
& \quad\quad (0.9)^t \\
& t \log_{0.9}\left(\frac{1}{1.2 * n}\right)
\end{aligned}
$$

The given schema disappears when,

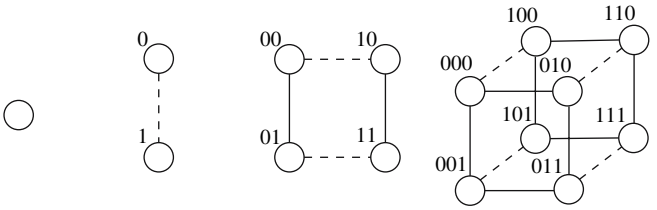
$$
\begin{aligned}
& n = 20 \Rightarrow t \geq 31 \\
& n = 50 \Rightarrow t \geq 39 \\
& n = 100 \Rightarrow t \geq 46 \\
& n = 200 \Rightarrow t \geq 53
\end{aligned}
$$

Illustrations on Search Space, hypercubes, schemata and Gray Codes

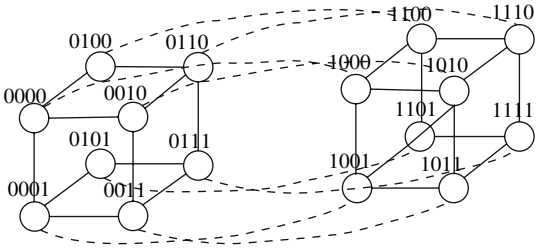
Illustration 5:

Suppose we deal with a 'Simple GA' which uses bit-strings of length L for chromosomes.

1. **Why is the representation of the search space of size 2^L by a hypercube more appropriate than a 'linear representation'**.



0–D Hypercube 1–D Hypercube 2–D Hypercube 3–D Hypercube



4–D Hypercube

Linear Representation
00...**00**|**00**...**01**|**00**...**10**|...|**01**...**11**|**10**...**00**|...

Solution:
Properties of hypercube:

(i) d-dimensional hypercube: 2 $(d-1)$-dimensional hypercubes
(ii) every node is connected to or has as neighbors all other nodes for which the bit-representation differs at only 1 position. (Hamming distance 1)

So, neighboring genotypes are represented as neighbors.

3.10 Solving a Problem: Genotype and Fitness

In order to apply a GA to a given problem the first decisions one has to make are concerning the kind of genotype the problem needs. That means that the user must decide how the parameters of the problem maps into a binary string (if a binary

string is to be used). This implies that the user must choose the number of bits per parameter, the range of the decoded binary-to-decimal parameter, if the strings have constant or changing lengths during the GA operations, etc...

In traditional applications the strings use a binary alphabet and their length is constant during all the evolution process. Also all the parameters decode to the same range of values and are allocated the same number of bits for the genes in the string. In most recent applications this situation is changing in a very important way. More and more complex applications need dynamic length strings or else a different non-binary alphabet. Typically integer or float genes are using in many domains as neural networks training, function optimization with a large number of variables, reordering problems as the Traveling Salesperson Problem, etc...

The traditional concept of a schema advises the utilization of binary genes because the number of schemata sampled per string is maximized. But a new interpretation of the schema concept has arrived to explain why many people have found very useful using other non-binary genes. This new interpretation considers beneficial, the utilization of higher cardinality alphabets due to their higher expression power (as many traditional AI paradigms consider). It is based on the consideration of the wildcard symbol (*) as an instantiation character for different sets of symbols and not the traditional wildcard symbol. To be precise, * is reinterpreted as a family of symbols ${}^{*}x$, where x can be any subset of symbols of the alphabet. For binary strings these two interpretations are the same but for higher cardinality alphabets they are not.

Regarding genotypes, many other developments have pushed forward the utilization of non-string representations such as trees of symbols that are much more appropriate for a large number of applications. Also extending every allele of the string along with the position (locus) it occupies in the string is interesting because the genes of the same string can be freely mixed in order to get together the building blocks automatically during evolution. This is very interesting when the building blocks of the problem are unknown and when the GA is required to discover them. By associating the locus to every gene value the string can be reordered in the correct way before evaluating the string. These mechanisms are common to the so-called messy GAs.

Here the user arrives to the problem of decoding (called *expression*) the genotype to yield the phenotype, that is, the string of problem parameters. This is necessary since the user needs to evaluate the string in order to assign it a fitness value. Thus the phenotype is used as the input to the fitness function and a fitness value that helps the algorithm to relatively rank the strings in the population pool is obtained.

The construction of an appropriate fitness function is very important for the correct work of the GA. This function represents the problem environment, that is, it decides which string solves the problem well. Many problems arise when the fitness function has to be constructed:

- The fitness function depends on whether the criterion is to be maximized or minimized.
- The environment can present noise in the evaluations (partial evaluation for example).

- The fitness function may change dynamically as the GA proceeds.
- The fitness function might be so complex that only approximations to fitness values can be computed.
- The fitness function should allocate very different values to strings in order to facilitate the work of the selection operator.
- The fitness function must consider the constraints of the problem. Normally if unfeasible solutions can appear the fitness function must allocate a small fitness value (if it is a maximization problem).
- The fitness function might incorporate some different sub-objectives. Such a multi-objective function presents non-conventional problems when used in the GA.
- The fitness function is a black box for the GA. It is fed with the phenotype and then the fitness value is obtained. Internally this may be achieved by a mathematical function, a complex computer simulator program or a human used that decides who good a string is (it is normal when strings encode images or music or something alike).

Regarding dynamic fitness functions a genotype of diploid genes can be used in which every value is determined by some relationship among two alleles encoded in the same string. The extension to K-ploid strings is straightforward. A dominance operator needs to be defined in order to get a single gene (parameter) value from the K alleles that determine it. As the evolution proceeds the fitness values assigned by the fitness function to the strings are very similar since the strings that are evaluated are also very similar. There exist some different scaling operators that help in separating the fitness values in order to improve the work of the selection mechanism. The most common ones are:

- Linear Scaling: $f' = af + b$
- Sigma Truncation: $f' = max[0, f - (f^* - c)]$, where f^* is the mean fitness value (standard deviation) in the population.
- Power Scaling: $f' = (f)^{\wedge}k$

Also when different strings get the same fitness values an important problem arises in that crossing them yield very bad individuals (called *lethals*) unable of further improvements. Sharing methods help to allow the concurrent existence of very different solutions in the same genetic population. Also parallel models help in overcoming this problem. Sometimes is very interesting to rank the population according to the fitness values of the strings and then to apply a reproductive plan that uses the rank position of the strings instead of the fitness values. This is good for ill-designed fitness function or when very similar fitness values are expected. This mechanism is called ranking, and keeps a constant selection pressure on the population.

3.10.1 Non-Conventional Genotypes

As mentioned before, genetic algorithms have traditionally used a pure binary genotype in which every parameter of the problem is encoded in binary by using a given

number of bits. The number of bits for every gene (parameter) and the decimal range in which they decode are usually the same but nothing precludes the utilization of a different number of bits or range for every gene. In order to avoid the so-called *hamming cliffs* the gray code is sometimes used when decoding a binary string. With this code any two adjacent decimal values are encoded with binary numbers that only differ in one bit. This helps the GA to make a gracefully approach to an optimum and normally reduces the complexity of the resulting search.

Nevertheless, the plethora of applications to which GAs have been faced have motivated the design of many different genotypes. This means that the string can be constructed over an integer or real alphabet, for example. Integer alphabets have interesting properties for reordering problems such as the Traveling Salesman Problem and others. On the other hand floating-point genes are very useful for numeric optimization (for example for neural network training). These higher cardinality alphabets enhance the search by reducing the number of ill-constructed strings resulting from the application of crossover or mutation. Also, when a very large number of parameters has to be optimized, it is helpful to rely on shorter strings (also because populations and the number of steps are smaller).

If the genotype can be expressed as a string, for any given application decision has to be made on whether the genome length is going to be kept constant or allowed to change. Also, it is an interesting decision to use diploid individuals in dynamic environments (i.e., environments in which the fitness function changes as the search proceeds). In a diploid individual two -homologous- chromosomes describe the same set of parameters. When decoding the string some kind of *dominance* operator need to be used in order to decide the value of the problem parameter (gene).

In many recent applications more sophisticated genotypes are appearing:

- One individual can be a tree of symbols as shown in Figure 3.5
- The individual is a combination of a string and a tree.
- Some parts of the string can evolve and some others don't in different moments of the search.
- The individual can be a matrix of symbols.

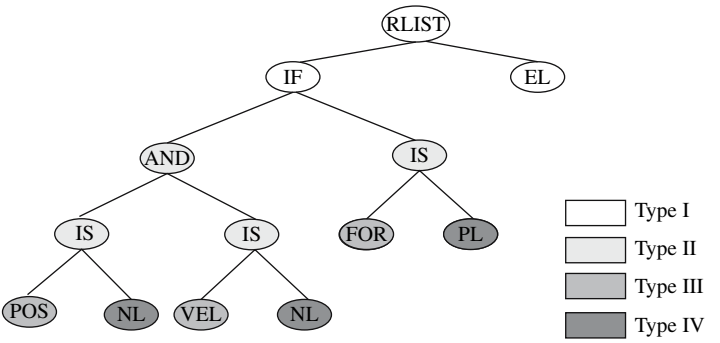


Fig. 3.5 Symbols tree

- The genotype could be a string of symbols that indicate operations to be performed. A simulator interprets the symbols and allocates fitness values depending on the result of the interpretation of the string.

3.11 Advanced Operators in GA

GA has a variety of operators other than the common crossover, mutation, reproduction etc., discussed in chapter 2. Some of the advanced operators are discussed in this section.

3.11.1 Inversion and Reordering

The order of genes on a chromosome is critical for the method to work effectively. Techniques for *reordering* the positions of genes in the chromosome during a run have been suggested. One such technique, *inversion*, works by reversing the order of genes between two randomly chosen positions within the chromosome. Reordering does nothing to lower epistasis, but greatly expands the search space. Not only is the GA trying to find *good sets of gene values*, it is simultaneously trying to discover *good gene orderings* too.

3.11.2 Epistasis

Epistasis is the interaction between different genes in a chromosome. It is the extent to which the "expression" (i.e. contribution to fitness) of one gene depends on the values of other genes. The degree of interaction will be different for each gene in a chromosome. If a small change is made to one gene a resultant change in chromosome fitness is observed. This resultant change may vary according to the values of other genes.

3.11.3 Deception

One of the fundamental principles of GAs is that chromosomes which include schemata which are contained in the global optimum will increase in frequency (this is especially true of short, low-order schemata, known as building blocks). Eventually, via the process of crossover, these optimal schemata will come together, and the globally optimum chromosome will be constructed. But if schemata which are not contained in the global optimum increase in frequency more rapidly than

those which are, the GA will be misled, away from the global optimum, instead of towards it. This is known as deception. Deception is a special case of epistasis and epistasis is necessary (but not sufficient) for deception. If epistasis is very high, the GA will not be effective. If it is very low, the GA will be outperformed by simpler techniques, such as hillclimbing.

3.11.4 Mutation and Naive Evolution

Mutation is traditionally seen as a "background" operator, responsible for re-introducing alleles or inadvertently lost gene values, preventing genetic drift and providing a small element of random search in the vicinity of the population when it has largely converged. It is generally held that crossover is the main force leading to a thorough search of the problem space. "Naive evolution" (selection and mutation) performs a hillclimb-like search which can be powerful without crossover. However, mutation generally finds better solutions than a crossover-only regime. Mutation becomes more productive, and crossover less productive, as the population converges. Despite its generally low probability of use, mutation is a very important operator. Its optimum probability is much more critical.

3.11.5 Niche and Speciation

Speciation is the process whereby a single species differentiates into two (or more) different species occupying different niches. In a GA, niches are analogous to maxima in the fitness function. If it is required to locate all the peaks with a multimodal fitness function, unfortunately a traditional GA will not do this; the whole population will eventually converge on a single peak. This is due to genetic drift. The two basic techniques to solve this problem are to maintain diversity, or to share the payoff associated with a niche. In pre-selection, offspring replace the parent only if the offspring's fitness exceeds that of the inferior parent. There is fierce competition between parents and children, so the payoff is not so much shared as fought over, and the winner takes all. This method helps to maintain diversity (since strings tend to replace others which are similar to themselves) and this helps prevent convergence on a single maximum. In a crowding scheme, offspring are compared with a few (typically 2 or 3) randomly-chosen individuals from the population. The offspring replaces the most similar one found. This again aids diversity and indirectly encourages speciation.

3.11.6 Restricted Mating

The purpose of restricted mating is to encourage speciation, and reduce the production of lethals. A lethal is a child of parents from two different niches. Although

each parent may be highly fit, the combination of their chromosomes may be highly unfit if it falls in the valley between the two maxima The general philosophy of restricted mating makes the assumption that if two similar parents (i.e. from the same niche) are mated, then the offspring will be similar. However, this will very much depend on the coding scheme and low epistasis. Under conventional crossover and mutation operators, two parents with similar genotypes will always produce offspring with similar genotypes. However, in a highly epistatic chromosome, there is no guarantee that these offspring will not be of low fitness, i.e. lethals. The total reward available in any niche is fixed, and is distributed using a bucket-brigade mechanism. In sharing, several individuals which occupy the same niche are made to share the fitness payoff among them. Once a niche has reached its "carrying capacity", it no longer appears rewarding in comparison with other, unfilled niches.

3.11.7 Diploidy and Dominance

In the higher lifeforms, chromosomes contain two sets of genes, rather than just one. This is diploidy. (A haploid chromosome contains only one set of genes.) Virtually all work on GAs concentrates on haploid chromosomes. This is primarily for simplicity, although use of diploid chromosomes might have benefits. Diploid chromosomes lend advantages to individuals where the environment may change over a period of time. Having two genes allows two different "solutions" to be remembered, and passed on to offspring. One of these will be dominant (that is, it will be expressed in the phenotype), while the other will be recessive. If environmental conditions change, the dominance can shift, so that the other gene is dominant. This shift can take place much more quickly than would be possible if evolutionary mechanisms had to alter the gene. This mechanism is ideal if the environment regularly switches between two states.

3.12 Important Issues in the Implementation of a GA

This section follows an initial summary of issues relating to the implementation of a GA:

- **Do not implement the population as an array of fixed length**: Implementation will not be flexible and sorting or insertions/replacements will be slow. Instead a list of pointers can be used.
- **Do not make re-evaluations of individuals when the fitness value is needed**. This error is very common when people develop general GA libraries. This might be negligible for toy applications but real applications do not admit two or more evaluations of the same individual (because of very high computational costs).

- **Do not implement in traditional imperative languages a library for evolutionary algorithms**. Reuse, error detection, comparisons and many other operations are readily easy if an object oriented language is used for the implementation.
- **Do not implement with a Logic Language**, the implementation requires a lot of memory and time of computation.

3.13 Comparison of GA with Other Methods

Any efficient optimization algorithm must use two techniques to find a global maximum: *exploration* to investigate new and unknown areas in the search space, and *exploitation* to make use of knowledge found at points previously visited to help find better points. These two requirements are contradictory, and a good search algorithm must find a tradeoff between the two. This section compares GA with other search methods.

3.13.1 Neural Nets

Both GAs and neural nets are adaptive, learn, can deal with highly nonlinear models and noisy data and are robust, "weak" random search methods. They do not need gradient information or smooth functions. In both cases their flexibility is also a drawback, since they have to be carefully structured and coded and are fairly application-specific. For practical purposes they appear to work best in combination: neural nets can be used as the prime modelling tool, with GAs used to optimize the network parameters.

3.13.2 Random Search

The brute force approach for difficult functions is a random, or an enumerated search. Points in the search space are selected randomly, or in some systematic way, and their fitness evaluated. This is a very unintelligent strategy, and is rarely used by itself.

3.13.3 Gradient Methods

A number of different methods for optimizing well-behaved continuous functions have been developed which rely on using information about the gradient of the

function to guide the direction of search. If the derivative of the function cannot be computed, because it is discontinuous, for example, these methods often fail. Such methods are generally referred to as *hillclimbing*. They can perform well on functions with only one peak (*unimodal* functions). But on functions with many peaks, (multimodal functions), they suffer from the problem that the first peak found will be climbed, and this may not be the highest peak. Having reached the top of a local maximum, no further progress can be made.

3.13.4 Iterated Search

Random search and gradient search may be combined to give an iterated hillclimbing search. Once one peak has been located, the hillclimb is started again, but with another, randomly chosen, starting point. This technique has the advantage of simplicity, and can perform well if the function does not have too many local maxima. However, since each random trial is carried out in isolation, no overall picture of the "shape" of the domain is obtained. As the random search progresses, it continues to allocate its trials evenly over the search space. This means that it will still evaluate just as many points in regions found to be of low fitness as in regions found to be of high fitness. A GA, by comparison, starts with an initial random population, and allocates increasing trials to regions of the search space found to have high fitness. This is a disadvantage if the maximum is in a small region, surrounded on all sides by regions of low fitness. This kind of function is difficult to optimize by any method, and here the simplicity of the iterated search usually wins.

3.13.5 Simulated Annealing

This is essentially a modified version of hill climbing. Starting from a random point in the search space, a random move is made. If this move takes us to a higher point, it is accepted. If it takes us to a lower point, it is accepted only with probability p(t), where t is time. The function p(t) begins close to 1, but gradually reduces towards zero, the analogy being with the cooling of a solid. Initially therefore, any moves are accepted, but as the "temperature" reduces, the probability of accepting a negative move is lowered. Negative moves are essential sometimes if local maxima are to be escaped, but obviously too many negative moves will simply lead us away from the maximum. Like the random search, however, simulated annealing only deals with one candidate solution at a time, and so does not build up an overall picture of the search space. No information is saved from previous moves to guide the selection of new moves.

3.14 Types of Genetic Algorithm

Genetic algorithms are a new generation for optimization and search techniques. They have become very popular because of their efficiency, power, utility, and accessibility. There are several types of genetic algorithms (GAs) designed to deal with various simple and complicated problem domains that range from an array of disciplines, including engineering, information technology, medicine, business, and the like. Due to the broad scope of the GAs and their applications, they have also been adapted to decipher and solve simple and complex GAs, including and obtaining optimal solutions to fuzzy genetic algorithms and multi modal, and multiple optimal solutions to an array of problems. This section discusses the types of GAs such as Sequential GA, Parallel GA, Messy GA, Hybrid GA, Adaptive GA, Generational GA and Steady state GA.

3.14.1 Sequential GA

The sequential genetic algorithm operates on a population of strings or, in general, structures of arbitrary complexity representing tentative solutions. In GA, every string is called an *individual* and it is composed of one or more chromosomes and a *fitness* value as shown in Figure 3.6. Normally, an individual contains just one chromosome that represents the set of parameters called *genes*. Every gene is in turn encoded in (usually) binary by using a given number of *alleles* (0,1). The evolutionary algorithm for a sequential GA is shown in Figure 3.7.

Sequential evolutionary algorithm

Sequential GAs have many successful applications to very different domains but there exists a number of drawbacks in their utilization, namely:

- They are usually inefficient.
- Evaluations are very time-consuming.
- Populations need to be large for a numerous number of problems.

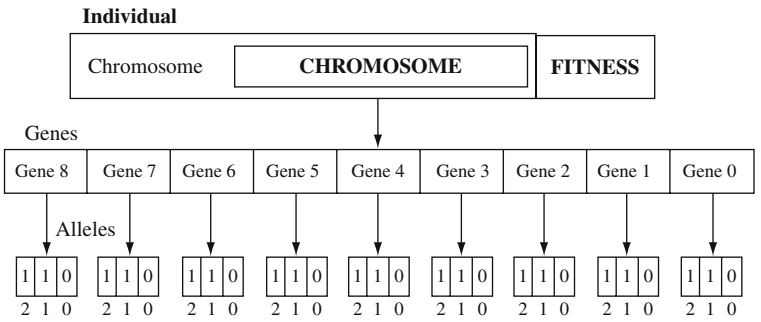


Fig. 3.6 Details on the genotype

```
t := 0;
initialize & evaluate [P(t)];
while not stop_condition do
          P'(t) := variation [P(t)];
          evaluate [P'(t)];
          P(t+1) := select [P'(t)];
          t := t +1;
end while
```

Fig. 3.7 Pseudo-code for a sequential evolutionary algorithm

3.14.2 Parallel GA

In order to solve the shortcomings of sequential GA and to study new models of higher efficiency and efficacy, parallel models of GAs have been devised. Parallel GAs are not only an extension of the traditional GA sequential model but they represent a new class of algorithms in that they make a different search work. Thus a **P**arallel **G**enetic **A**lgorithm (PGA) is not only expected to be more efficient but also to need a smaller number of evaluations of the target function. Many different approaches to PGAs exist and they differ in many aspects.

When referring to a parallel genetic algorithm it is important to distinguish between PGA as a particular model of a GA and a PGA as a means of implementing a GA. In a PGA model, the full population exists in a distributed form; either multiple independent subpopulations exist, or there is one population but each population member interacts only with a limited set of neighbors.

One advantage of the PGA model is that traditional GA tend to convergence prematurely, an effect that PGAs seem to be able to partially mitigate because of their ability to maintain more diverse subpopulations by exchanging genetic material between subpopulations. Also, in a traditional GA the expected number of offspring of a string depends on the strings's fitness relative to all other strings in the population. This situation implies a global ranking that is unlike the way natural selection works.

Many GA researchers believe a PGA is a more realistic model of species in nature than a single large population; by analogy with natural selection, a population is typically many independent subpopulations that occasionally interact. PGAs also naturally fit the model of the way evolution is viewed as occurring; a large degree of independence exists in the global population.

A parallel implementation of the traditional sequential GA model is also possible. A simple way to do this is to parallelize the loop that creates the next generation from the previous one. Most of the steps in this loop (evaluation, crossover, mutation and local search if used) can be executed in parallel. The selection step, depending on the selection algorithm, may require a global sum that can be a parallel bottleneck. When such an approach has been taken, it is often on a distributed memory computer. However, unless function evaluation is a time consuming step, the parallel computing overheads associated with distributing data structures to processors and

synchronizing and collecting the results, can mitigate any performance improvements due to multiple processors.

Pseudo-Code of PGA

As a stochastic technique, in a PGA, the three major steps are initial sampling, optimization and checking the stopping criterion. Therefore, it begins $(t = 0)$ by randomly creating a population $P\ (t = 0)$ of m structures, each one encoding the p problem variables on some alphabet. Each structure is usually a vector and each problem variable is encoded in l_x bits or in a floating-point number. However, as mentioned before, other non-string representations are possible. The pseudo code of PGA is shown in Figure 3.8.

Advantages of PGA

Some of the advantages of parallel genetic algorithm are

- Works on a coding of the problem (least restrictive)
- Basically independent of the problem (robustness)
- Can yield alternative solutions to the problem
- Parallel search from multiple points in the space
- Easy parallelization as islands or neighborhoods
- Better search, even if no parallel hardware is used
- Higher efficiency and efficacy than sequential GAs
- Easy cooperation with other search procedures

$t := 0;$
initialize: $P(0) := \{\vec{a}_1(0),\ldots,\vec{a}_\mu(0)\} \in I^\mu;$
evaluate: $P(0) : \{\Phi(\vec{a}_1(0)),\ldots,\Phi(\vec{a}_\mu(0))\};$
while not $(P(t))$ **do** *// Reproductive Loop*
select: $P'(t) := s_{\Theta_s}(P(t));$
recombine: $P''(t) := \otimes_{\Theta_r}(P'(t));$
mutate: $P'''(t) := m_{\Theta_m}(P''(t));$
evaluate: $P'''(t) : \{\Phi(\vec{a}_1'''(t)),\ldots,\Phi(\vec{a}_\lambda'''(t))\};$
replace: $P(t+1) := r_{\Theta_r}(P'''(t) \cup Q);$
<communication step> SYNC ASYNC
$t := t+1;$
end while

Fig. 3.8 Pseudo-code for a parallel genetic algorithm

3.14.3 Hybrid GA

Unlike other search and optimization techniques, a genetic algorithm promises convergence but not optimality, not even that it will find local maxima. This implies that the choice of when to stop the genetic algorithm is not well-defined. The genetic algorithm process is stopped when 50 generations have gone by with no better chromosome identified. Since there is no guarantee of optimality, successive runs of the GA will provide different chromosomes with varying fitness measures. This is one of the drawbacks of using a genetic algorithm for optimization - since there is no guarantee of optimality, there is always the chance that there is a better chromosome lurking somewhere in the search space.

Although there is no guarantee of optimality, exponential convergence is assured. If the GA is run several times, it will converge each time, possibly at different optimal chromosomes. The schemata which promise convergence are actually indicative of the regions in the search space where good chromosomes may be found. Typically, the GA is coupled with a local search mechanism to find the optimal chromosome in a region. So, if a hybrid algorithm is used, the problem reduces to ensuring that the GA is run as many times as is needed to pick out all the good regions. If the shape of the search space is known previously, the number of regions can be easily estimated. Then the GA can be run repeatedly until these regions have been found. In most practical problems, however, the shape of the search space is not known before hand. The systematic approach is then to repeat GA runs until the best chromosomes that are found to repeat with some regularity.

GAs are not good at identifying the optimal value of a chromosome for a problem but do very well in identifying the regions where those optima lie. Therefore, a hybrid GA is used - every ten generations, the user anneals the best 10% of the population. This has the effect of moving the top chromosomes in that generation (which are the result of exponential convergence toward the best regions) to the local maximum in their region.

A Hybrid Genetic Algorithm is designed to use heuristics for improvement of offspring produced by crossover. Initial population is randomly generated. The offspring is obtained by crossover between two parents selected randomly. The layout improvement heuristics RemoveSharp and LocalOpt are used to bring the offspring to a local maximum. If fitness of the layout of the offspring thus obtained is greater than the fitness of the layout of any one of the parents then the parent with lower fitness is removed from the population and the offspring is added to the population. If the fitness of the layout of the offspring is lesser than that of both of its parent then it is discarded. For mutation a random number is generated within one and if it is less than the specified probability of the mutation operator a layout is randomly selected and removed from the population. Its layout is randomized and then added to the population. The algorithm works as below:

Step 1:

Initialize population randomly

Step 2:

Apply RemoveSharp algorithm to all layouts in the initial population
Apply LocalOpt algorithm to all layouts in the initial population

Step 3:

Select two parents randomly
Apply Crossover between parents and generate an offspring
Apply RemoveSharp algorithm to offspring
Apply LocalOpt algorithm to offspring
If Fitness(offspring) > Fitness (any one of the parents) then replace the weaker parent by the offspring

Step 4:

Mutate any one randomly selected layout from population

Step 5:

Repeat steps 3 and 4 until end of specified number of iterations.

Thus a hybrid algorithm ensures that the GA is run as many times as is needed to pick out all the good regions and uses heuristics for improvement of offspring produced by crossover.

3.14.4 Adaptive GA

Adaptive genetic algorithms (AGA) are GAs whose parameters, such as the population size, the crossing over probability, or the mutation probability are varied while the GA is running. A simple variant could be the following: The mutation rate is changed according to changes in the population; the longer the population does not improve, the higher the mutation rate is chosen. Vice versa, it is decreased again as soon as an improvement of the population occurs.

Adaptive Probabilities of Crossover and Mutation

It is essential to have two characteristics in GAs for optimizing multimodal functions. The first characteristic is the capacity to converge to an optimum (local or global) after locating the region containing the optimum. The second characteristic is the capacity to explore new regions of the solution space in search of the global optimum. The balance between these characteristics of the GA is dictated by the values of mutation probability (p_m) and crossover probability (p_c), and the type of crossover employed. Increasing values of p_m and p_c promote exploration at the expenssse of exploitation. Moderately large values of p_c (0.5–1.0) and small values of p_m (0.001-0.05) are commonly employed in GA practice. In this approach, the

trade-off between exploration and exploitation is achieved in a different manner, by varying p_c and p_m adaptively in response to the fitness values of the solutions; p_c and p_m are increased when the population tends to get stuck at a local optimum and are decreased when the population is scattered in the solution space.

Design of Adaptive p_c and p_m

To vary p_c and p_m adaptively for preventing premature convergence of the GA to a local optimum, it is essential to be able to identify whether the GA is converging to an optimum. One possible way of detecting is to observe average fitness value $\bar{f}$ of the population in relation to the maximum fitness value $f_{\max}$ of the population. $f_{\max} - \bar{f}$ is likely to be less for a population that has converged to an optimum solution than that for a population scattered in the solution space. It is found that $f_{\max} - \bar{f}$ decreases when the GA converges to a local optimum with a fitness value of 0.5 (The globally optimal solution has a fitness value of 1.0).The difference in the average and maximum fitness values, $f_{\max} - \bar{f}$, is used as a yardstick for detecting the convergence of the GA. The values of p_c and p_m are varied depending on the value of $f_{\max} - \bar{f}$. Since p_c and p_m have to be increased when the GA converges to a local optimum, i.e., when $f_{\max} - \bar{f}$ decreases, p_c and p_m will have to be varied inversely with $f_{\max} - \bar{f}$. The expressions that are chosen for p_c and p_m are of the form

$$p_c = k_1/(f\max - \bar{f}) \; and$$
$$p_m = k_2/(f\max - \bar{f}).$$

It has to be observed in the above expressions that p_c and p_m do not depend on the fitness value of any particular solution, and have the same values for all the solution of the population. Consequently, solutions with high fitness values as well as solutions with low fitness values are subjected to the same levels of mutation and crossover. When a population converges to a globally optimal solution (or even a locally optimal solution), p_c and p_m increase and may cause the disruption of the near-optimal solutions. The population may never converge to the global optimum. Though GA is prevented from getting stuck at a local optimum, the performance of the GA (in terms of the generations required for convergence) will certainly deteriorate.

To overcome the above stated problem, 'good' solutions of the population have to be preserved. This can be achieved by having lower values of p_c and p_m for high fitness solutions and higher values of p_c and p_m for low fitness solutions. While the high fitness solutions aid in the convergence of the GA, the low fitness solutions prevent the GA from getting stuck at a local optimum. The value of p_m should depend not only on $f_{\max} - \bar{f}$, but also on the fitness value f of the solution. Similarly, p_c should depend on the fitness values of both the parent solutions. The closer f is to $f_{\max}$ the smaller p_m should be, i.e., p_m should vary directly as $f_{\max} - f$. Similarly, p_c should vary directly as $f_{\max} - f'$, where f' is the larger of the fitness value of the

solutions to be crossed. The expressions for p_c and p_m now take the forms.

$$p_c = k_1((f_{max}–f')/(f_{max} - \bar{f}), \quad k_1 \leq 1.0$$
$$p_m = k_2((f_{max}–f)/(f_{max} - \bar{f})), \quad k_2 \leq 1.0$$

(k_1 and k_2 have to be less than 1.0 to constrain p_c and p_m to the range 0.0–1.0).

It is found that p_c and p_m that are zero for the solution with the maximum fitness. Also $p_c = k_1$ for a solution with $f' = \bar{f}$, and $p_m = k_2$ for a solution with $f = \bar{f}$. For solution with sub average fitness values i.e., $f < \bar{f}$, p_c and p_m might assume values larger than 1.0. To prevent the overshooting of p_c and p_m beyond 1.0, the following constraints are available,

$$p_c = k_{3,} \quad f' \leq \bar{f}$$
$$p_m = k_{4,} \quad f \leq \bar{f}$$

where $k_3, \ k_4, \leq 1.0$.

Practical Considerations and Choice of Values for k_1, k_2, k_3 and k_4

In the previous subsection, it was found that for a solution with the maximum fitness value p_c and p_m are both zero. The best solution in a population is transferred undisrupted into the next generation. Together with the selection mechanism, this may lead to an exponential growth of the solution in the population and may cause premature convergence. To overcome the above stated problem, a default mutation rate (of 0.005) is introduced for every solution in the AGA.

he choice of values for $k_1, \ k_2, \ k_3$ and k_4 is discussed. For convenience, the expressions for p_c and p_m are given as

$$p_c = k_1(f_{\max} - f')/(f_{\max} - \bar{f}), \quad f' \geq \bar{f}$$
$$p_c = k_{3,} \qquad\qquad\qquad\qquad f' < \bar{f}$$

and

$$p_m = k_2(f_{\max} - f')/(f_{\max} - \bar{f}), \quad f \geq \bar{f}$$
$$p_m = k_{4,} \qquad\qquad\qquad\qquad f < \bar{f}$$

Where $k_1, \ k_2, \ k_3, \ k_4 \leq 1.0$.

It has been well established in GA literature that moderately large values of p_c $(0.5 < p_c < 1.0)$, and small values of p_m $(0.001 < p_m < 0.05)$ are essential for the successful working of GAS. The moderately large values of p_c promote the extensive recombination of schemata, while small values of p_m are necessary to prevent the disruption of the solutions. These guidelines, however, are useful and relevant when the values of p_c and p_m not vary.

One of the goals of the approach is to prevent the GA from getting stuck at a local optimum. To achieve this goal, solutions are employed with sub average fitness's to search the search space for the region containing the global optimum. Such solutions need to be completely disrupted, and for this purpose a value of 0.5 is used for k_4. Since solutions with a fitness value of $\bar{f}$ should also be disrupted completely, a value of 0.5 is assigned to k_2 as well.

Based on similar reasoning, k_1 and k_3 are assigned a value of 1.0. This ensures that all solutions with a fitness value less than or equal to $\bar{f}$ compulsorily undergo crossover. The probability of crossover decreases as the fitness value (maximum of the fitness values of the parent solutions) tends to $f_{\max}$ and is 0.0 for solutions with a fitness value equal to $f_{\max}$. Next, a comparison is performed on the AGA with previous approaches for employing adaptive operators in GAs.

In the adaptive GA, low values of p_c and p_m are assigned high fitness solutions, while low fitness solutions have very high values of p_c and p_m. The best solution of every population is 'protected' i.e. it is not subjected to crossover, and receives only a minimal amount of mutation. On the other hand, all solution with a fitness value less than the average fitness value of the population have $p_m = 0.5$. This means that all sub average solutions are completely disrupted and totally new solutions are created. The GA can thus rarely get stuck at a local optimum.

3.14.5 Integrated Adaptive GA (IAGA)

It is typical for real-world genetic algorithms to define several types of genetic operators, whether unary, binary or multi-ary. In general, the rates of the operators do not have optimal values which are valid for an entire run of the genetic algorithm. More often than not, a strategy for optimally setting the rates is hard to find by the designer of the genetic algorithm. On the other hand, each type of operator has parameters which control the algorithm it defines. Finding the appropriate values for these parameters is a difficult task, as well.

IAGA is a general schema of a genetic algorithm which is able to dynamically adapt both the rate and the behavior for each of its operators. The rates of the operators are adapted in a deterministic, reinforcement-based manner. The behavior of each operator (that is, the specific way it operates) is modified by changing its parameter values. Operators that prove themselves valuable during the optimization process are rewarded; thus, they are applied more often in subsequent generations. The behavior of each operators evolves as well during the run of the genetic algorithm, by modifying the values of the parameters which control its activity.

3.14.6 Messy GA

Motivation for the messy GA arises from viewing the objective for fitness function *f(x)* as the sum of *m* independent sub functions $f_i(x_i)$ each defined on the same number of loci *k* where *k* is the estimated level of deception present in the most deceptive sub function. A building block is a set of genes which fully specify the independent variable of some sub function. A highly fit building block (HFBB) is defined to be the building block which optimizes the corresponding sub function. Thus the (unique) string containing only HFBB's optimizes the objective function. The messy GA consists of

- Initialization
- Primordial
- Juxtapositional phase

Its improved ability to solve deceptive problems stems from the focus on increasing the proportion of HFBB's in the population before applying recombinative operators.

Original Messy GA

Messy makes use of Partially Enumerative Initialization (PEI) which results in a population consisting of all possible partial solutions defined over *k* loci. Thus each building block is represented exactly once, although not every string contains a building block since the loci over which a string is defined may correspond to different sub functions. For an application in which each string contains ℓ genes, and each gene has *C* possible alleles, the initial population contains

$$N = k^C(\frac{l}{k})$$

solutions. For even modest values of *k* this is significantly larger than a typical simple GA population size. For example, for a problem using a binary representation with $\ell = 50$ and $k = 5$, the initial population contains 6.78×10^7 individuals. Typical simple GA population sizes are in the range of tens to thousands.

Tournament selection is then used in the primordial phase to reduce the population size by eliminating less fit partial solutions. Competition is limited to those partial solutions for which the number of common defining loci is greater than the expected value. The shuffle size parameter specifies the maximum number of individuals examined in searching for a compatible mate. A locally optimal solution, called the competitive template, is used to "fill in the gaps" in partially specified solutions to allow their evaluation and subsequent selection.

The juxtapositional phase is similar to the simple GA in its use of recombinative operators. Standard crossover is replaced by cut-and-splice, which is a one-point crossover operating on variable length strings. Splice and bitwise cut probabilities are specified, and are normally chosen to promote rapid string growth from *k* to ℓ.

Fast Messy GA

Fast messy genetic algorithm is a special clone of common simple genetic algorithm (GA). This type represents new, more powerful kind in the GA branch. It resists premature local-minimum fall and solves problems in shorter time. A gene is represented by a pair *(allele locus, allele value)* in messy algorithms, so that – for instance – chromosome (2,0),(0,1),(1,1) represents 3 bit-long chromosome 110. Operation cut and splice are used to bread new offspring.

Incomplete specification of the chromosome (under specification) or redundant specification (over specification) could occur during evolution. Over specification is solved by right-to-left scan, i.e. the only first occurrence of an appropriate pair is taken into account. For example: ((0,0), (2,1), (1,0), (2,0)) represents chromosome 001. On the other hand, under specification is harder to deal with. Hence a template is used to represent a solution of the chromosome which is shown in Figure 3.9.

In order to reduce the size of the messy GA initial population and the execution time of the initialization and primordial phases three modifications were proposed to the original algorithm:

– use of Probabilistically Complete Initialization (PCI) in place of PEI
– use of building block filtering
– more conservative thresholding in tournament selection

The three phases of each iteration of operation is illustrated in Figure 3.10. The objective of PCI is to ensure that each HFBB has an expected number of copies in the initial population sufficient to overcome sampling noise. Each individual in the PCI population is defined at $\ell' = \ell - k$ loci, which are selected randomly without replacement (it is assumed that $k \ll \ell$). After accounting for noise, the required population size is

$$N = \frac{\left| \begin{matrix} \ell \\ \ell' \end{matrix} \right|}{\left| \begin{matrix} \ell - k \\ \ell' - k \end{matrix} \right|} 2z_\alpha^2 \beta^2 (m-1) 2^k$$

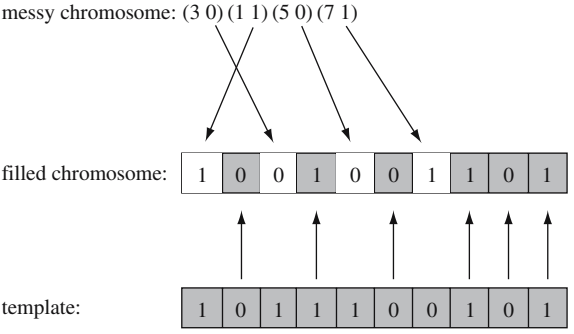


Fig. 3.9 Representation of a solution - chromosome

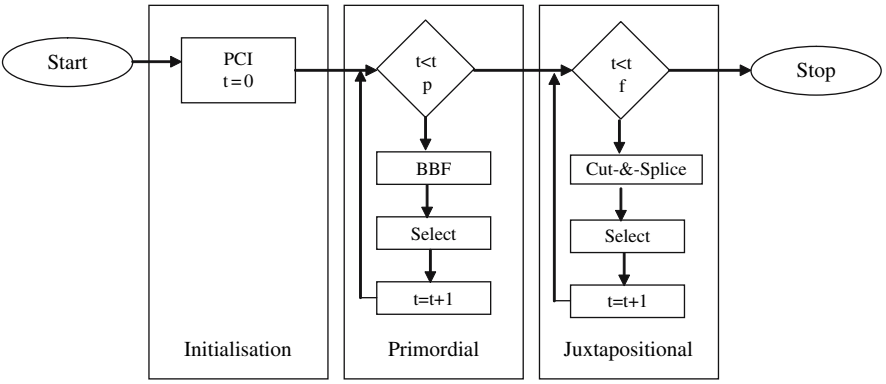


Fig. 3.10 Flow chart of fast messy genetic algorithm

where ℓ, ℓ', k and m have been defined previously, α is a parameter specifying the probability of selection error between two competing building blocks, $P[Z \geq z_\alpha] = 1 - \alpha$ where Z is a standard normal random variable and β^2 is a parameter specifying the maximum inverse signal-to-noise ratio per subfunction to be detected.

The fast messy GA uses tournament selection and building block filtering (BBF) to enrich the population in the primordial phase. The effect of several iterations of tournament selection is to eliminate nearly all of the individuals in the population which contain fewer HFBB's, while increasing the number of copies of individuals which contain more. BBF then reduces the number of defining loci for each individual by randomly deleting some number of genes. In the process it disrupts many but not all of the HFBB's. Some individuals containing HFBB's remain to receive additional copies in subsequent iterations of tournament selection. For any particular problem, some HFBB's may contribute more to the total fitness of the optimal solution than others. Also, a small number of individuals may contain multiple HFBB's. Since all of the HFBB's must be represented in the population if there is to be any chance of combining them, competition is restricted to those individuals which contain building blocks corresponding to the same subfunction. Theoretically, this may be achieved by allowing competition only between those individuals which share at least

$$\theta = \left[\frac{\lambda^2}{\ell} + z_\alpha^2 \sigma \right]$$

common defining loci, where λ is the number of defining loci for the individuals, and

$$\sigma^2 = \frac{\lambda^2(\ell - \lambda)^2}{\ell^2(\ell - 1)}$$

is the variance of the distribution of a random variable L corresponding to the number of defining loci shared by two randomly selected individuals. Thus, θ is calculated using a normal distribution approximating the actual distribution of L in the initial population. The approximation ignores the dependence of L's distribution on the dynamics of the genetic population in subsequent generations. By carefully choosing a primordial schedule of tournament selection and BBF, a population can be generated which consists of strings of length k and which is dominated by HFBB's. In current practice, it is common to use a full shuffle and an empirically determined threshold schedule.

3.14.7 Generational GA (GGA)

The generational genetic algorithm creates new offspring from the members of an old population using the genetic operators and places these individuals in a new population which becomes the old population when the whole new population is created. Usually, in generational replacement based GAs the whole population is replaced in every iteration.

In GGA, the entire population is replaced each generation by their offspring. The hope is that the offspring of the best strings carry the important "building blocks" from the best strings forward to the next generation. The GGA allows the possibility that the best strings in the population do not survive to the next generation. Many of the best strings may not be allocated any reproductive trials. It is also possible that mutation or crossover destroy or alter important bit values so that they are not propagated into the next generation by the parent's offspring. Many implementations of the GGA use elitism: if the best string in the old population is not chosen for inclusion in the new population, it is included in the new population anyway. The idea is to avoid losing the best string found so far.

Figure 3.11 represents the generational GA. Here $P(t)$ is the population of strings at generation t. For each generation selection, recombination and evaluation takes place.

```
t ← 0
initialize P(t)
for each generation
      t ← t + 1
      select P(t + 1) from P(t)
      recombine P(t + 1) from P(t)
      evaluate P(t + 1)
end for
```

Fig. 3.11 Generational Genetic Algorithm

3.14.8 Steady State GA (SSGA)

The incremental/steady state genetic algorithm is different to the generational model in that there is typically one single new member inserted into the new population at any one time. A replacement/deletion strategy defines which member of the population will be replaced by the new offspring. On the other hand, in steady state GAs only a few individuals are replaced in each iteration. SSGAs having smaller population sizes normally lose diversity very fast, and larger population sizes increase the cost of computation and slow down the speed of convergence. In SSGA since only one or two individuals are replaced in each iteration, it can provide a better scope to keep the population distributed over a larger fraction of search space by *appropriately choosing individuals* undergoing genetic operations and individuals which are to be deleted. This, in turn, will increase the expectation of maintaining diversity in the population, adopt to changes in the objective function by reallocating future search effort toward the region favored by the present environment.

The SSGA is an alternative to the GGA that replaces only a few individuals at a time, rather than the entire generation. In practice, the number of new strings to create each generation is usually one or two. The new strings replace the worst ranked strings in the population. In this way SSGA allows both parents and their offspring to co exist in the same population.

The SSGA has a built in elitism since only the lowest ranked string is deleted; the best string is automatically kept in the population. Also, the SSGA is immediately able to take advantage of the genetic material in a newly generated string without having to wait to generate the rest of the population as in the GGA.

A disadvantage of the SSGA is that with small populations some bit positions are more likely to lose their value than with a GGA. For this reason SSGAs' are often run with large population sizes to offset this.

Figure 3.12 presents the steady state genetic algorithm. Here $P(t)$ is the population of strings at generation t. At each generation one new string is inserted into the population. The first step is to pick a random string, x_{random}, and apply a local search heuristic to it. Next two parent strings x_1 and x_2 are selected, and a random number, $r \in [0,1]$, is generated. If r is less than the crossover probability p_c, then two new offsprings are created through crossover and one of them x_{new} is randomly selected to insert in the population. If r is not less than the crossover probability then one of the two parent strings is randomly selected, a copy of that parent is made, and mutation is applied to flip bits in the copy with probability 1/n. In either case, the new string is tested to see whether it duplicates a string already in the population. If it does, it undergoes mutation until it is unique. The least-fit string in the population is deleted x_{new}, is inserted and the population is reevaluated.

```
        t ← 0
initialize P(t)
evaluate P(t)
for each generation
        local_search (x_random ∈ P(t))
        select (x1, x2) from P(t)
        if (r < pc) then
            x_new = crossover (x1, x2)
        else
            x_new = mutate (x1, x2)
        endif
        delete (x_worst ∈ P(t))
        while ()(x_new ∈ P(t))
            mutate(x_new)
        P(t + 1) ← P(t) x_new
        Evaluate P(t + 1)
        t ← t + 1
end for
```

Fig. 3.12 Steady-state genetic algorithm

3.15 Advantages of GA

Some of the advantages of GA are:

- GA can quickly scan a vast solution set. Bad proposals do not effect the end solution negatively as they are simply discarded.
- The inductive nature of the GA means that it doesn't have to know any rules of the problem - it works by its own internal rules. This is very useful for complex or loosely defined problems.
- They efficiently search the model space, so they are more likely (than local optimization techniques) to converge toward a global minima.
- There is no need of linearization of the problem.
- There is no need to compute partial derivatives.
- More probable models are sampled more frequently than less probable ones.

Perhaps it isn't obvious why such an algorithm should lead to accurate solutions for optimisation problems. Crossover is a crucial aspect of any genetic algorithm, but it may seem that it will dramatically change parents with a high fitness function so that they will no longer be fit. However this is not the case. As in biology, crossover can lead to new combinations of genes which are more fit than any in the previous generations. Other offspring will be created which are less fit but these will have a low probability of being selected to go on to the next generation. Creating new variants is the key to genetic algorithms, as there is a good chance of finding better solutions. This is why mutation is also a necessary part of the program. It will create offspring which would not have arisen otherwise, and may lead to a better solution.

Other optimisation algorithms have the disadvantage that some kind of initial guess is required and this may bias the final result. GAs on the other hand only require a search range, which need only be constrained by prior knowledge of the physical properties of the system. Effectively they search the whole of the solution space, without calculating the fitness function at every point. This can help avoid a danger in any optimisation problem: being trapped in local maxima or minima. There are two main reasons for this:

i) the initial population, being randomly generated, will sample the whole of the solution space, and not just a small area;
ii) variation-inducing tactics, ie crossover and mutation, prevent the algorithm being trapped in one part of the solution space.

3.16 Matlab Examples of Genetic Algorithms

3.16.1 Genetic Algorithm Operations Implemented in MATLAB

****

Reproduction

In reproduction, offspring are bred by the selected individuals. For generating new chromosomes, the algorithm can use both recombination and mutation. The following code snippet is used to implement the reproduction function in MATLAB.

```
function result=reproduce(population)
% Returns next generation of a population

fitted_pop=distribution(population);
for i=1:(size(population,1)/2)
    x=select(fitted_pop);
    y=select(fitted_pop);
    [x,y]=crossover(x,y);
    result(2*i-1,:)=x;
    result(2*i,:)=y;
end
```

****

Selection

The selection operator is intended to improve the average quality of the population by giving the high-quality chromosomes a better chance to get copied into the next generation. It is defined as the ratio of the probability of selection of the best

chromosome in the population to that of an average chromosome. Hence, a high selection pressure results in the population's reaching equilibrium very quickly, but it inevitably sacrifices genetic diversity. The following code snippet is used to implement the selection function in MATLAB.

```
function result=select(popWithDistrib)
% Select some genotypes from the population,
% and possibly mutates them.

selector=rand;
total_prob=0;
% Default to last in case of rounding error
genotype=popWithDistrib(end,2:end);
for i=1:size(popWithDistrib,1)
    total_prob=total_prob+popWithDistrib(i,1);
    if total_prob > selector
      genotype=popWithDistrib(i,2:end);
      break;
    end
end
result=mutate(genotype);
```

****

Crossover

Crossover examines the current solutions in order to find better ones. Physically, crossover in the Shortest Path problem plays the role of exchanging each partial route of two chosen chromosomes in such a manner that the offspring produced by the crossover represents only one route. The crossover between two dominant parents chosen by the selection gives higher probability of producing offspring having dominant traits. The following code snippet is used to implement the crossover function in MATLAB.

```
function [x,y]=crossover(x,y)
% Possibly takes some information from one genotype and
% swaps it with information from another genotype

if rand < 0.6
    gene_length=size(x,2);
    % site is between 2 and gene_length
    site=ceil(rand * (gene_length-1))+1;
    tmp=x(site:gene_length);
    x(site:gene_length)=y(site:gene_length);
    y(site:gene_length)=tmp;
end
```

****

Fitness Function

A fitness function is a particular type of objective function that quantifies the optimality of a solution (that is, a chromosome) in a genetic algorithm so that, that particular chromosome may be ranked against all the other chromosomes. The fitness function interprets the chromosome in terms of physical representation and evaluates its fitness based on traits of being desired in the solution. Optimal chromosomes, or at least chromosomes which are more optimal, are allowed to breed and mix their datasets by any of several techniques, producing a new generation that will (hopefully) be even better. The following code snippet is used to retrieve the fitness function in MATLAB.

```
function result=fitness(population);
% Returns the fitness for each row in a population

result=sum(population, 2);

*****************************************************
function result=distribution(population)
% Takes the population data and returns thepopulation
% data with each genotype paired with its fraction of
% the total fitness of the population

genotypes=noduplicates(population);
total_fitness=sum(fitness(genotypes));
result=[(fitness(genotypes)/total_fitness),genotypes];
*****************************************************
```

Mutation

In genetic algorithms, mutation is a genetic operator used to maintain genetic diversity from one generation of a population of chromosomes to the next. It is analogous to biological mutation. The classic example of a mutation operator involves a probability that an arbitrary bit in a genetic sequence will be changed from its original state. A common method of implementing the mutation operator involves generating a random variable for each bit in a sequence. This random variable tells whether or not a particular bit will be modified. The purpose of mutation in GAs is to allow the algorithm to avoid local minima by preventing the population of chromosomes from becoming too similar to each other, thus slowing or even stopping evolution. This reasoning also explains the fact that most GA systems avoid only taking the fittest of the population in generating the next but rather a random (or semi-random) selection with a weighting toward those that are fitter. The following code snippet is used to mutate a given genotype in MATLAB.

```
function result=mutate(genotype)
% Possibly mutates a genotype
result=abs(genotype - rand(size(genotype,1),
size(genotype,2))<0.03));
*******************************************************
```

3.16.2 Illustration 1 – Maximizing the given Function

Maximize the function $\mathbf{f(x) = x + 10^{*}sin(5^{*}x) + 7^{*}cos(4^{*}x)}$ **over the interval (0,9)** and plot the end population

Solution:

The basic genetic algorithm is used to maximize the given function. To refresh, the algorithm is as follows:

Step 1. Create an initial population (usually a randomly generated string)
Step 2. Evaluate all of the individuals (apply some function or formula to the individuals)
Step 3. Select a new population from the old population based on the fitness of the individuals as given by the evaluation function.
Step 4. Apply genetic operators (mutation & crossover) to members of the population to create new solutions.
Step 5. Evaluate these newly created individuals.
Step 6. Repeat steps 3–6 (one generation) until the termination criteria has been satisfied (usually perform for a certain fixed number of generations.

MATLAB CODE:

As an initial process the given function $\mathbf{f(x) = x + 10^{*}sin(5^{*}x) + 7^{*}cos(4^{*}x)}$ **over the interval (0,9)** is plotted using the following code. The plot is shown in Figure 3.13.

```
fplot('x+10*sin(5*x)+7*cos(4*x)',[0 9])
```

Now, a genetic algorithm is set up to find the maximum of this problem. First, the evaluation function is created in terms of a .m file. The function used here is gaDemoEval.m

```
function [sol, val]=gaDemo1Eval(sol,options)
x=sol(1);
val=x+10*sin(5*x)+7*cos(4*x);
```

The evaluation function must take two parameters, sol and options. Sol is a row vector of n + 1 elements where the first n elements are the parameters of interest. The

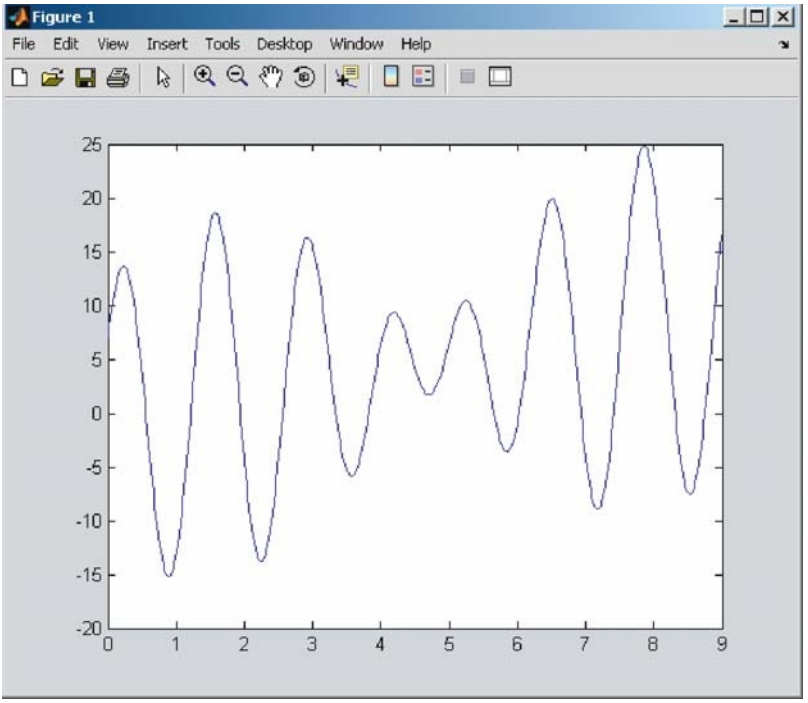


Fig. 3.13 Plot of the given function

$(n+1)$th element is the value of this solution. The options matrix is a row matrix of [current generation, eval options].

The eval function must return both the value of the string, val and the string itself, sol. This is done so that the evaluation can repair and/or improve the string.

Now that the evaluation function is defined, an initial population has to be created. The most common way to generate an initial population is to randomly generate solutions within the range of interest, in this case 0–9.

The following initializega routine will do this initialization process.

```
function [pop]=initializega(populationSize, variable
Bounds,evalFN, evalOps,options)

% initializega creates a matrix of random numbers with
% a number of rows equal
% to the populationSize and a number columns equal to
% the number of rows in
% bounds plus 1 for the f(x) value which is found by
% applying the evalFN.
% This is used by the ga to create the population if
```

```
  % it is not supplied.
  % pop - the initial, evaluated, random population
  % populationSize - the size of the population, i.e.
  % the number to create
  % variableBounds - a matrix which contains the bounds
  % of each variable, i.e.
  % [var1_high var1_low; var2_high var2_low; ....]
  % evalFN - the evaluation fn, usually the name of the
  % .m file for evaluation
  % evalOps - any options to be passed to the eval
  % function defaults []
  % options - options to the initialize function, ie.
  % [type prec] where
  % eps is the epsilon
  % value and the second option is
  % 1 for float and 0 for
  % binary, prec is the precision of the
  % variables defaults [1e-6 1]

  Let's create a random starting population of size 10.

  initPop=initializega(10,[0 9],'gademo1eval1');
```

To have a look at this population the initPop is plotted as shown in Figure 3.14.

```
  hold on
  plot (initPop(:,1),initPop(:,2),'g+')
  As a next step the evolutionary procedure is run.
  function [x,endPop,bPop,traceInfo]=ga(bounds,evalFN,
                                        evalOps,startPop,
                                        opts,termFN,termOps,
                                        selectFN,selectOps,
                                        xOverFNs,xOverOps,
                                        mutFNs,mutOps)
% Output Arguments:
  % x             - the best solution found during the
  %                   course of the run
  % endPop        - the final population
  % bPop          - a trace of the best population
  % traceInfo     - a matrix of best and means of the ga
  %                   for each generation

% Input Arguments:
  % bounds        - a matrix of upper and lower bounds
  %                   on the variables
```

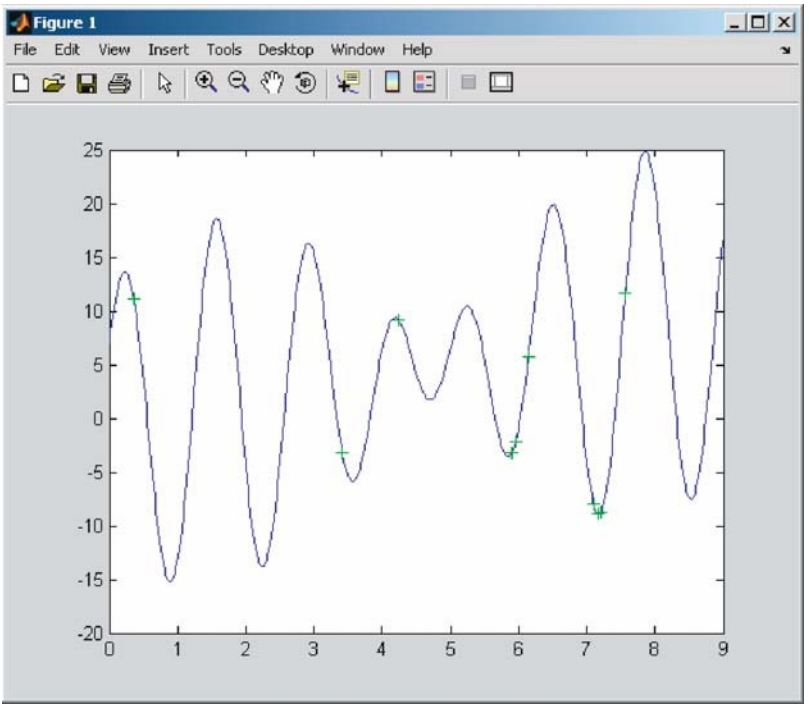


Fig. 3.14 Plot of the initial population of the given function (marked in green)

```
% evalFN        - the name of the evaluation .m
%                  function
% evalOps       - options to pass to the evaluation
%                  function ([NULL])
% startPop      - a matrix of solutions that can be
%                    initialized from initialize.m
% opts          - [epsilon prob_ops display] change
%                  required to consider two
%                  solutions different, prob_ops 0 if
%                  the user wants to apply the genetic
%                  operators probabilistically to each
%                  solution, 1 if the user is supplying
%                  a deterministic number of operator
%                  applications and display is 1 to
%                  output progress 0.
% termFN        - name of the .m termination function
%                  (['maxGenTerm'])
% termOps       - options string to be passed to the
%                  termination function ([100]).
% selectFN      - name of the .m selection function
```

```
%                  (['normGeomSelect'])
% selectOpts     - options string to be passed to
%                  select after select select(pop,
%                  #,opts) ([0.08])
% xOverFNS - a string containing blank separated names
%                   of Xover.m
% files (['arithXover heuristicXover simpleXover'])
% xOverOps - A matrix of options to pass to Xover.m
%                  files with the first column being
%                  the number of that xOver to perform
% similarly for mutation ([2 0;2 3;2 0])
%  mutFNs - a string containing blank separated names
%           of mutation.m % files (['boundaryMutation
%           multi NonUnifMutation ...
%           nonUnifMutation unifMutation'])
%           mutOps - A matrix of options to pass to
%           Xover.m files with the
%           first column being the number of that
%           xOver to perform
%           similarly for mutation ([4 0 0;6 100 3;4
%           100 3;4 0 0])
```

Now the GA is run for one generation.

```
[x endPop]=ga([0 9],'gademo1eval1',[],initPop,
  [1e-6 11],'maxGenTerm',1,... 'normGeomSelect',[0.08],
  ['arithXover'],[2 0],'nonUnifMutation',[2 1 3]);
```

The best found values after one generation is given as

x - The best found

```
x=
    7.5648 11.6257
```

The resulting population is plotted and the plot is shown in Figure 3.15

```
plot (endPop(:,1),endPop(:,2),'ro')
```

Next the GA is run for 25 generations

```
[x endPop]=ga([0 9],'gademo1eval1',[],initPop,
[1e-6 1 1],
'maxGenTerm',25,...
'normGeomSelect',[0.08],['arithXover'],[2],
'nonUnifMutation',[2 25 3]);
```

The output of the GA after 25 generations is shown below:

```
Generation 1 - 11.625663
Generation 2 - 0
Generation 3 - 0
Generation 4 - 0
Generation 5 - 17.056272
Generation 6 - 0
Generation 7 - 19.722312
Generation 8 - 24.759537
```

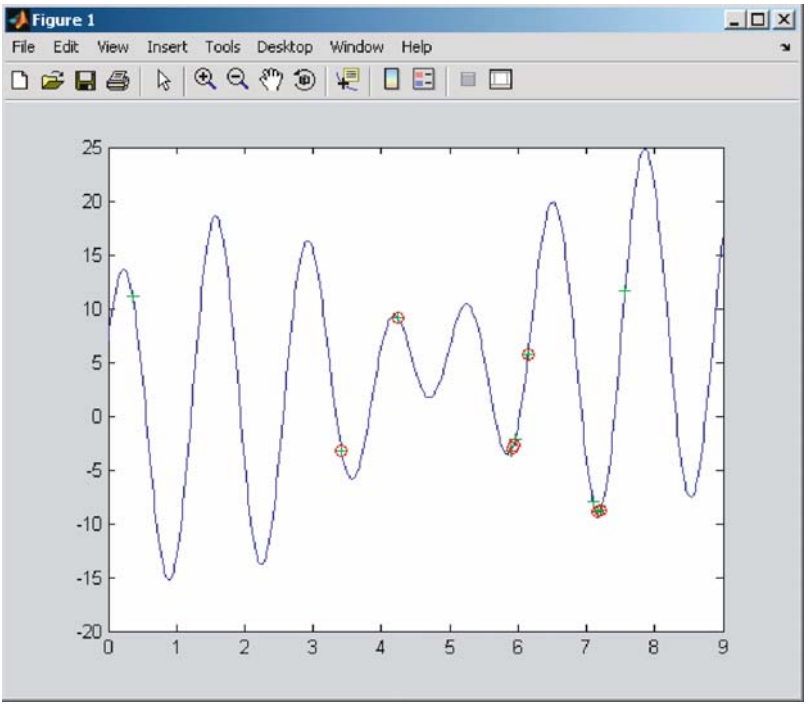


Fig. 3.15 Plot of the end population (marked as red circles)

```
Generation 9 - 0
Generation 10 - 0
Generation 11 - 0
Generation 12 - 0
Generation 13 - 0
Generation 14 - 24.794904
Generation 15 - 24.814713
Generation 16 - 0
Generation 17 - 24.824588
Generation 18 - 24.844660
Generation 19 - 24.855255
Generation 20 - 0
Generation 21 - 0
Generation 22 - 0
Generation 23 - 0
Generation 24 - 24.855306
Generation 25 - 24.855306
```

The best found value of x is

```
x=
    7.8573 24.8553
```

The resulting population is plotted and the plot is shown in Figure 3.16.

```
plot (endPop(:,1),endPop(:,2),'y*')
```

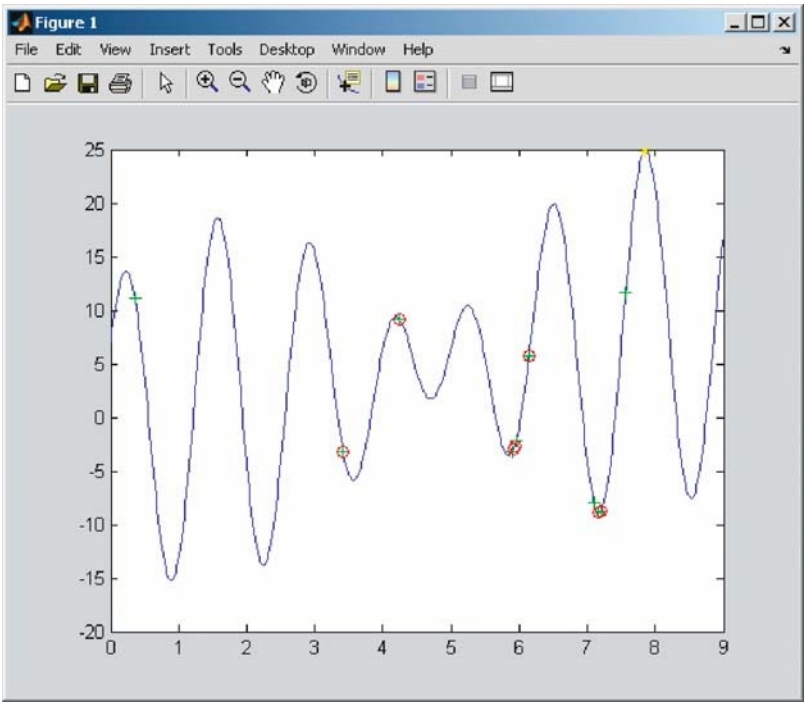


Fig. 3.16 Plot of the end population (marked as yellow stars)

3.16.3 Illustration 2 – Optimization of a Multidimensional Non-Convex Function

Using the genetic toolbox optimize a multidimensional non-convex function

```
% This demonstration show the use of the genetic
% toolbox to
% optimize a multi- & dimensional non-convex function.
function [val]=coranaEval(sol)
% Determines the value of the Corana function at
% point sol.
% val - the value of the Corana function at point sol
% sol - the location to evaluate the Corana function
numv=size(sol,2);
x=sol(1:numv);
d0=[1 1000 10 100 1 10 100 1000 1 10];
d=d0(1:numv);
c=0.15;
s=.2*ones(1,numv);
```

```
t=0.05*ones(1,numv);
bk=s.*(round(x./s));
dev=(abs(bk-x)<t) & (bk~=0);
z=c*((bk+sign(bk).*t).^2).*d;
y=x.^2.*d;
val=sum((dev.*z)+((~dev).*y));
% This function is basically a n dimensional parabola
% with rectangular
% pockets removed.
% Let's take a look at the function in
% 2-dimensions
echo off
% First consider it in each dimension independently
clf
plot(z(:,1))
%Plot a slice of the function in x max 250.25
 (Figure 3.17)
```

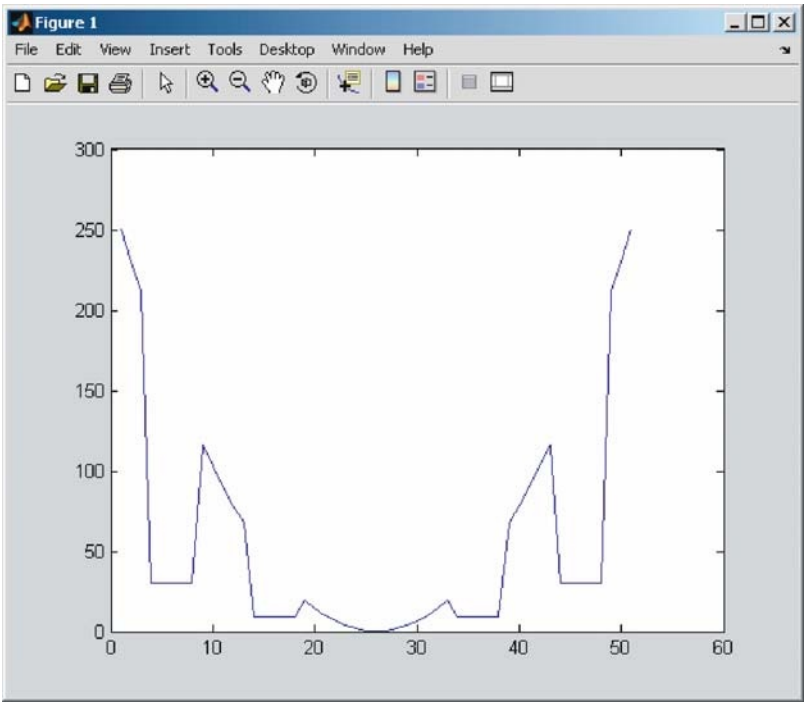


Fig. 3.17 Plot of slice of the function in x

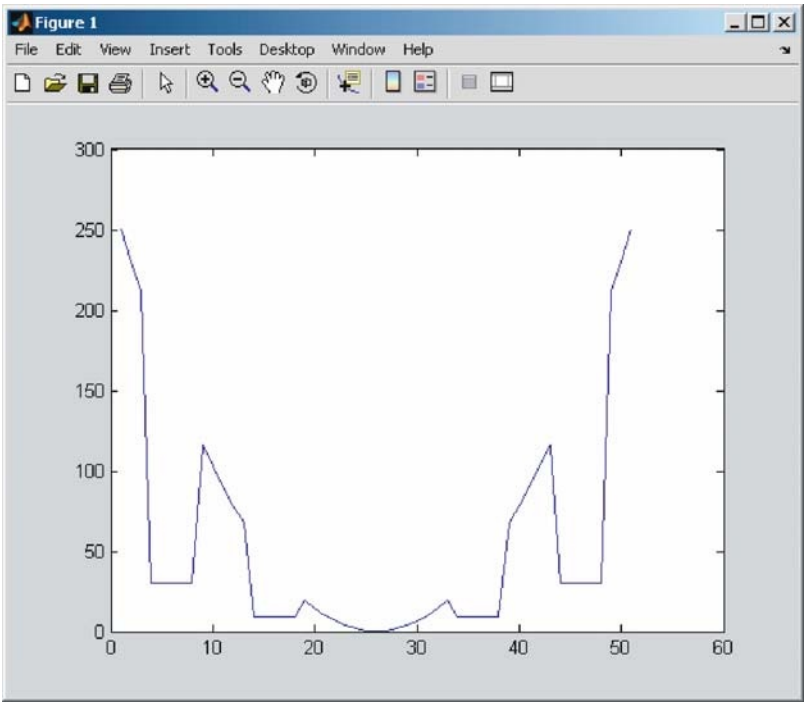


Fig. 3.18 Plot of slice of the function in y

```
% Notice that the range is [250.0-250.25]
clf
plot(z(1,:))
% Plot a slice of the function in y (Figure 3.18)
% Notice the range is [0-250]
pause
% Strike any key to continue
mesh(a,a,z);
view(30,60);
grid;
% Remember the deviation in y is 1000 times that of x.
% Lets minimize this function in 4 dimensions between
% [-10,000 10,000].
% The ga is set up to maximize only. Minimization of
% f(x) is
% equivalent to
% maximizing -f(x), so negative of the Corana function
% is used.
```

```
function [sol,val]=coranaMin(sol,options)
% Function to minimize the Corana function.
%
% val - the value of the Corana function at point sol
% sol - the location to evaluate the Corana function
numVar=size(sol,2)-1;
val=coranaEval(sol(1:numVar));
val=-val;
% First the bounds are set up
bounds=ones(4,1)*[-10000 10000];
% Then the function is optimized as
[x,endPop,bestSols,trace]=ga(bounds,'coranaMin');
% The first return is the optimal [x1 x2 x3 x4 val]
```

Result
$x = [6.1763 \; -0.0069 \quad 0.3313 \quad -0.0244 \quad -7.0643]$

```
% The performance of the GA during the run is plotted
% as follows and the plot
% is shown in Figure 3.19
plot(trace(:,1),trace(:,3),'b-')
```

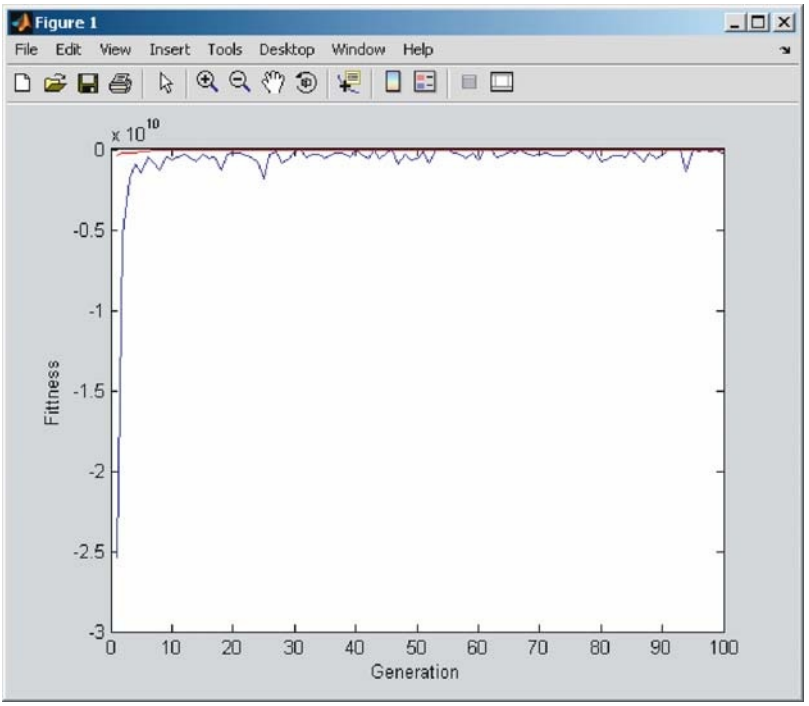


Fig. 3.19 Performance of GA - The red line is a track of the best solution, the blue is a track of the average of the population

```
hold on
plot(trace(:,1),trace(:,2),'r-')
xlabel('Generation'); ylabel('Fittness');
```

3.16.4 Illustration 3 – Traveling Salesman Problem

Find an optimal solution to the Traveling Salesman Problem by using GA to determine the shortest path that is required to travel between cities.

Solution:

The MATLAB function tsp_ga takes input argument as the number of cities. Then the distance matrix is constructed and plotted. Then shortest path is then obtained using GA as an optimization tool.

MATLAB Code:

The MATLAB code used to implement the traveling sales man problem is shown below:

```
function varargout=tsp_ga(varargin)
% the function tsp_ga finds a (near) optimal solution
% to the Traveling
% Salesman Problem by setting up a Genetic Algorithm
% (GA) to search for the
% shortest path (least distance needed to travel to
% each city exactly once)
```

The input argument to the MATLAB function tsp_ga can be any one of the following:

- tsp_ga(NUM_CITIES) where NUM_CITIES is an integer representing the number of cities (default= 50)

  For example tsp_ga(25) solves the TSP for 25 random cities
- tsp_ga(CITIES) where CITIES is an N × 2 matrix representing the X/Y coordinates of user specified cities

  For example tsp_ga(10*RAND(30,2)) solves the TSP for the 30 random cities in the 10*RAND(30,2)) matrix
- tsp_ga(..., OPTIONS) or tsp_ga(OPTIONS) where OPTIONS include one or more of the following in any order:
  - '-NOPLOT' turns off the plot showing the progress of the GA
  - -RESULTS' turns on the plot showing the final results as well as the following parameter pairs:
  - 'POPSIZE', VAL sets the number of citizens in the GA population VAL should be a positive integer (divisible by 4) - - default= 100
  - 'MRATE', VAL sets the mutation rate for the GA VAL should be a float between 0 and 1, inclusive - - default= 0.85
  - 'NUMITER', VAL sets the number of iterations (generations) for the GA VAL should be a positive integer - - default= 500

– For example, the following code solves the TSP for 20 random cities using a population size of 60, a 75% mutation rate, and 250 GA iterations

```
tsp_ga(20, 'popsize', 60, 'mrate', 0.75, 'numiter',
250);
```

The following code solves the TSP for 30 random cities without the progress plot

```
[sorted_cities, best_route, distance]=tsp_ga(30,
'-noplot');
```

The following code solves the TSP for 40 random cities using 1000 GA iterations and plots the results

```
cities=10*rand(40, 2);
[sorted_cities]=tsp_ga(cities, 'numiter', 1000,
'-results');
```

Declare the parameters used in TSP

```
error(nargchk(0, 9, nargin));
num_cities=50;
cities=10*rand(num_cities, 2);
pop_size=100;
num_iter=500;
mutate_rate=0.85;
show_progress=1;
show_results=0;
```

Declare the Process Inputs

```
cities_flag=0;
option_flag=0;
for var=varargin
  if option_flag
    if ~ isfloat(var{1}), error(['Invalid value for
    option'
        upper(option)]);
    end
    switch option
        case 'popsize', pop_size=4*ceil(real(var{1}
        (1))/4);
               option_flag=0;
        case 'mrate', mutate_rate=min(abs(real(var{1}
        (1))), 1);
               option_flag=0;
        case 'numiter', num_iter=round(real(var{1}
        (1)));
               option_flag=0;
```

```
        otherwise, error(['Invalid option '
         upper(option)])
      end
    elseif ischar(var{1})
      switch lower(var{1})
        case '-noplot', show_progress=0;
        case '-results', show_results=1;
        otherwise, option=lower(var{1}); option_flag=1;
      end
    elseif isfloat(var{1})
      if cities_flag, error('CITIES or NUM_CITIES may be
              specified, but not both');
      end
      if length(var{1})==1
        num_cities=round(real(var{1}));
        if num_cities < 2, error('NUM_CITIES must be an
        integer
    greater
            than 1');
        end
        cities=10*rand(num_cities, 2); cities_flag=1;
      else
        cities=real(var{1});
        [num_cities, nc]=size(cities); cities_flag=1;
        if or(num_cities < 2, nc ~=2)
          error('CITIES must be an Nx2 matrix of floats,
with
          N > 1')
        end
      end
    else
      error('Invalid input argument.')
    end
  end
```

Construction of the Distance Matrix by using Distance measurement formula

```
  dist_matx=zeros(num_cities);
  for ii=2:num_cities
    for jj=1:ii-1
      dist_matx(ii, jj)=sqrt(sum((cities(ii, :)-cities(jj,
:)).
        ^2));
      dist_matx(jj, ii)=dist_matx(ii, jj);
  end
```

```
end
```

The cities and the distance matrix are plotted. The plot is shown in Figure 3.20.

```
if show_progress
  figure(1)
  subplot(2, 2, 1)
  plot(cities(:,1), cities(:,2), 'b.')
  if num_cities < 75
    for c=1:num_cities
        text(cities(c, 1), cities(c, 2),
        [' ' num2str(c)],
        'Color',
          'k', 'FontWeight', 'b')
    end
  end
  title([num2str(num_cities) ' Cities'])
  subplot(2, 2, 2)
  imagesc(dist_matx)
```

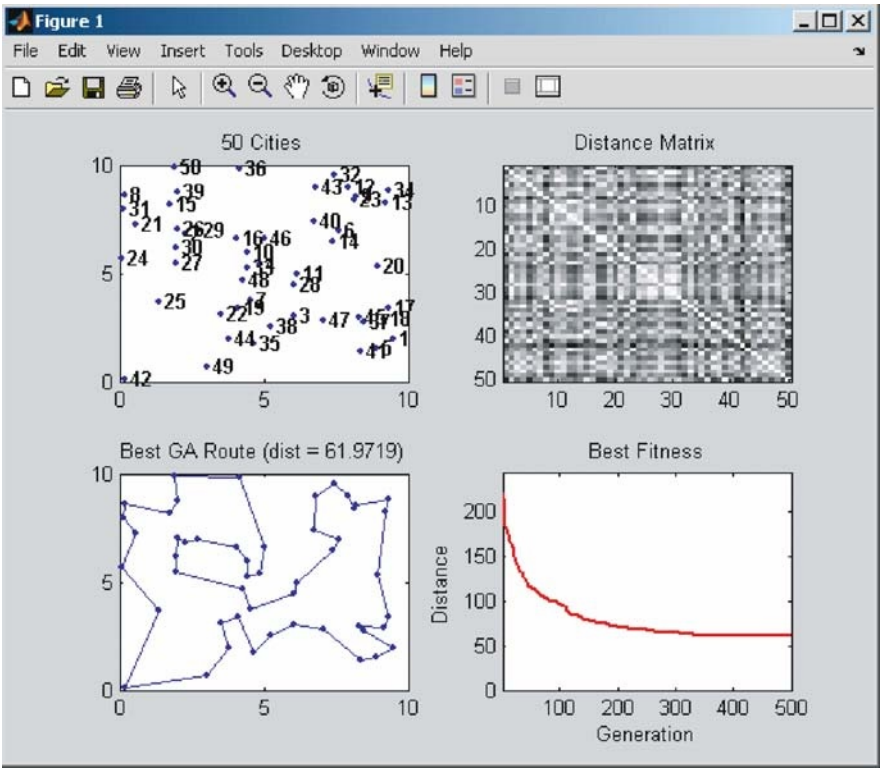


Fig. 3.20 Plot of cities, distance matrix, best GA route and best fitness for tsp_ga code

```
  title('Distance Matrix')
  colormap(flipud(gray))
end
```

Initialize Population in a random manner

```
pop=zeros(pop_size, num_cities);
pop(1, :)=(1:num_cities);
  for k=2:pop_size
pop(k, :)=randperm(num_cities);
end
```

Calculation of the best route

```
if num_cities < 25, display_rate=1; else
 display_rate=10; end
fitness=zeros(1, pop_size);
best_fitness=zeros(1, num_iter);
for iter=1:num_iter
  for p=1:pop_size
    d=dist_matx(pop(p, 1), pop(p, num_cities));
    for city=2:num_cities
       d=d+dist_matx(pop(p, city-1), pop(p, city));
    end
    fitness(p)=d;
  end
  [best_fitness(iter) index]=min(fitness);
  best_route=pop(index, :);
```

The best GA route gcalculated from the previous step is computed and plotted.

```
  if and(show_progress, ~mod(iter, display_rate))
    figure(1)
    subplot(2, 2, 3)
    route=cities([best_route best_route(1)], :);
    plot(route(:, 1), route(:, 2)', 'b.-')
    title(['Best GA Route (dist='
    num2str(best_fitness(iter))
       ')'])
    subplot(2, 2, 4)
    plot(best_fitness(1:iter), 'r', 'LineWidth', 2)
    axis([1 max(2, iter) 0 max(best_fitness)*1.1])
  end
% Genetic Algorithm Search
  pop=iteretic_algorithm(pop, fitness, mutate_rate);
end
% Plotting the best fitness. The plot is shown in
% Figure 3.20
```

```
if show_progress
  figure(1)
  subplot(2, 2, 3)
  route=cities([best_route best_route(1)], :);
  plot(route(:, 1), route(:, 2)', 'b.-')
  title(['Best GA Route (dist=' num2str(best_fitness
    (iter)) ')'])
  subplot(2, 2, 4)
  plot(best_fitness(1:iter), 'r', 'LineWidth', 2)
  title('Best Fitness')
  xlabel('Generation')
  ylabel('Distance')
  axis([1 max(2, iter) 0 max(best_fitness)*1.1])
end
if show_results
  figure(2)
  imagesc(dist_matx)
  title('Distance Matrix')
  colormap(flipud(gray))
  figure(3)
  plot(best_fitness(1:iter), 'r', 'LineWidth', 2)
  title('Best Fitness')
  xlabel('Generation')
  ylabel('Distance')
  axis([1 max(2, iter) 0 max(best_fitness)*1.1])
  figure(4)
  route=cities([best_route best_route(1)], :);
  plot(route(:, 1), route(:, 2)', 'b.-')
  for c=1:num_cities
    text(cities(c, 1), cities(c, 2), [' ' num2str(c)],
       'Color', 'k', 'FontWeight', 'b')
  end
  title(['Best GA Route (dist=' num2str(best_fitness
     (iter)) ')'])
end
[not_used indx]=min(best_route);
best_ga_route=[best_route(indx:num_cities) best_route(1:
  indx-1)];
if best_ga_route(2) > best_ga_route(num_cities)
  best_ga_route(2:num_cities)=fliplr(best_ga_route(2:
  num_cities));
end
varargout{1}=cities(best_ga_route, :);
varargout{2}=best_ga_route;
```

```
varargout{3}=best_fitness(iter);
```

The genetic algorithm search function is implemented using the MATLAB Code shown below:

```
function new_pop=iteretic_algorithm(pop, fitness,
 mutate_rate)
[p, n]=size(pop);
```

Tournament Selection - Round One

```
new_pop=zeros(p, n);
ts_r1=randperm(p);
winners_r1=zeros(p/2, n);
tmp_fitness=zeros(1, p/2);
for i=2:2:p
  if fitness(ts_r1(i-1)) > fitness(ts_r1(i))
    winners_r1(i/2, :)=pop(ts_r1(i), :);
    tmp_fitness(i/2)=fitness(ts_r1(i));
  else
    winners_r1(i/2, :)=pop(ts_r1(i-1), :);
    tmp_fitness(i/2)=fitness(ts_r1(i-1));
  end
end
```

Tournament Selection - Round Two

```
ts_r2=randperm(p/2);
winners=zeros(p/4, n);
for i=2:2:p/2
  if tmp_fitness(ts_r2(i-1)) > tmp_fitness(ts_r2(i))
    winners(i/2, :)=winners_r1(ts_r2(i), :);
  else
    winners(i/2, :)=winners_r1(ts_r2(i-1), :);
  end
end
new_pop(1:p/4, :)=winners;
new_pop(p/2+1:3*p/4, :)=winners;
```

Crossover

```
crossover=randperm(p/2);
children=zeros(p/4, n);
for i=2:2:p/2
  parent1=winners_r1(crossover(i-1), :);
  parent2=winners_r1(crossover(i), :);
  child=parent2;
  ndx=ceil(n*sort(rand(1, 2)));
  while ndx(1)==ndx(2)
    ndx=ceil(n*sort(rand(1, 2)));
  end
```

```
    tmp=parent1(ndx(1):ndx(2));
    for j=1:length(tmp)
      child(find(child==tmp(j)))=0;
    end
    child=[child(1:ndx(1)) tmp child(ndx(1)+1:n)];
    child=nonzeros(child)';
    children(i/2, :)=child;
  end
  new_pop(p/4+1:p/2, :)=children;
  new_pop(3*p/4+1:p, :)=children;
```

Mutate

```
  mutate=randperm(p/2);
  num_mutate=round(mutate_rate*p/2);
  for i=1:num_mutate
    ndx=ceil(n*sort(rand(1, 2)));
    while ndx(1)==ndx(2)
      ndx=ceil(n*sort(rand(1, 2)));
    end
    new_pop(p/2+mutate(i), ndx(1):ndx(2))=...
      fliplr(new_pop(p/2+mutate(i), ndx(1):ndx(2)));
  end
```

Output:

Thus the traveling salesman problem was executed in MATLAB and the results are plotted indicating the number of cities, Distance Matrix, the best GA route and the best fitness.

3.16.5 Illustration 4 – GA using Float Representation

Illustrate the use of GA using float representation.

Solution:

The problem is solved using a MATLAB Code as follows. The results are also plotted.

MATLAB Code:

This code shows the usage of GA using a float representation.

```
global bounds
```

Setting the seed to the same for binary

```
rand('seed',156789)
```

Crossover Operators

```
xFns='arithXover heuristicXover simpleXover';
xOpts=[1 0; 1 3; 1 0];
```

Mutation Operators

```
mFns='boundaryMutation multiNonUnifMutation
  nonUnifMutation unifMutation';
```

```
mOpts=[2 0 0;3 200 3;2 200 3;2 0 0];
```
Termination Operators
```
termFns='maxGenTerm';
termOps=[200]; % 200 Generations
```
Selection Function
```
selectFn='normGeomSelect';
selectOps=[0.08];
```
Evaluation Function
```
evalFn='gaEval';
evalOps=[];
type gaEval
```
Bounds on the variables
```
bounds=[-3 12.1; 4.1 5.8];
```
GA Options [epsilon float/binary display]
```
gaOpts=[1e-6 1 1];
```
Generate an intialize population of size 20
```
startPop=initializega(20,bounds,'gaEval',[1e-6 1])
```
Lets run the GA
```
[x endPop bestPop trace]=ga(bounds,evalFn,evalOps,
 startPop,
  gaOpts,...
termFns,termOps,selectFn,selectOps,xFns,xOpts,mFns,
 mOpts);
```
Here x is the best solution found, endPop is the ending population, bestPop is the best solution tracked over generations, trace is a trace of the best value and average value of generations. The best over time is plotted as shown in Figure 3.21.
```
clf
plot(trace(:,1),trace(:,2));
```
Add the average to the graph and plot the results. The plot is shown in Figure 3.22
```
hold on
plot(trace(:,1),trace(:,3));
```
The population size is increased by running the defaults
```
[x endPop bestPop trace]=ga(bounds,evalFn,evalOps,[],
gaOpts);
```
Here x is the best solution found, endPop is the ending population, bestPop is the best solution tracked over generations, trace is a trace of the best value and average value of generations. The best over time is plotted as shown in Figure 3.23
```
clf
plot(trace(:,1),trace(:,2));
```
Add the average to the graph and plot the results. The plot is shown in Figure 3.24
```
hold on
plot(trace(:,1),trace(:,3));
```

Output:
The start population of the given problem is

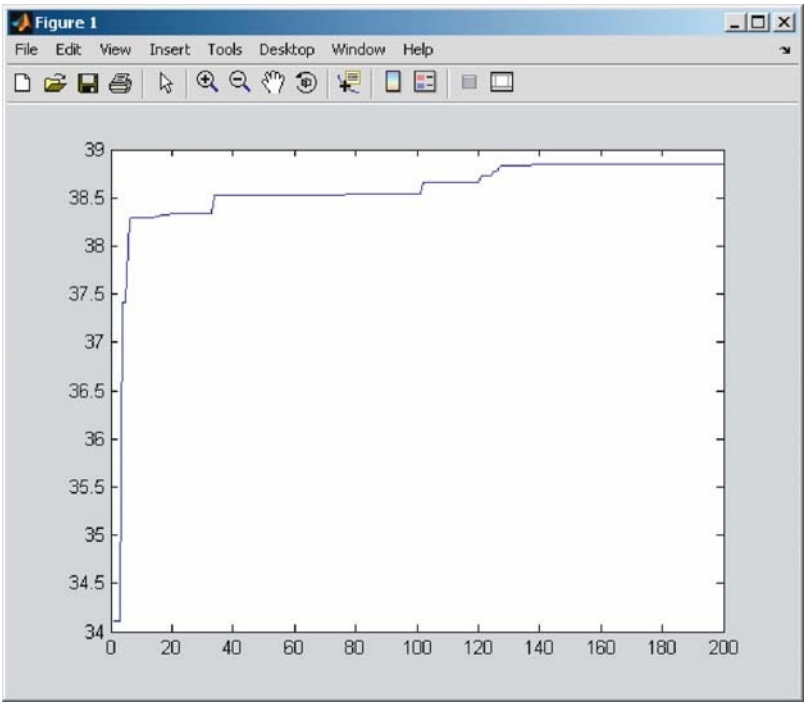


Fig. 3.21 Plot of the best over time

```
startPop=
4.2347   4.2633   19.1466
4.9077   5.0506   16.8107
6.4984   4.2130   24.4514
0.1479   5.0688   16.9514
8.8349   4.3104   16.3785
9.6764   5.7708   23.6559
3.7818   4.3481   20.5341
3.9687   4.5768   15.4287
3.5781   5.2348   28.7510
7.4248   5.7222   21.1179
1.5030   4.1321   25.2849
12.0997  4.5040   34.1072
6.6956   5.7131   29.9267
6.7457   4.4351   25.4436
3.2365   4.9655   17.9328
4.1093   4.4547   24.2278
−2.6759  5.7013   24.1061
8.7395   4.1341   26.1237
6.1001   4.7011   27.6249
9.6483   5.1743   25.5700
```

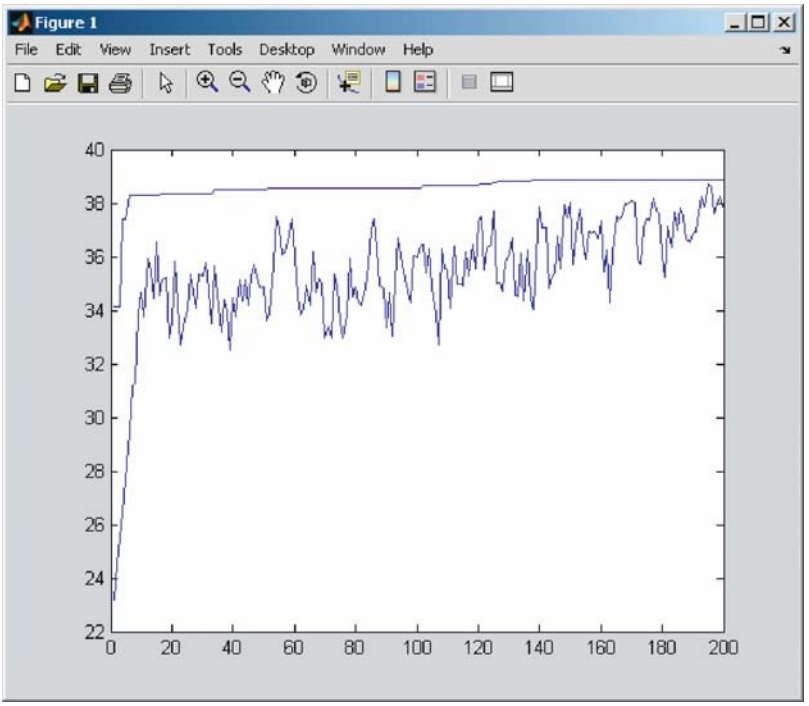


Fig. 3.22 Plot of the trace

The best found x =

```
11.6255   5.7250   38.8503
```

Population at the end of 200 generations is

```
endPop=
11.6255   5.7250   38.8503
11.6255   5.7250   38.8503
11.6255   5.7250   38.8503
−3.0000   5.7250   27.2250
11.6255   5.1412   35.8149
12.1000   5.7250   38.7328
11.6255   5.7250   38.8503
11.6255   5.7250   38.8503
11.6255   5.7250   38.8503
11.6255   4.2836   29.4455
11.6255   4.8424   35.3568
11.6255   5.7250   38.8503
11.6255   5.7250   38.8503
11.6255   5.7250   38.8503
11.6255   5.7250   38.8503
```

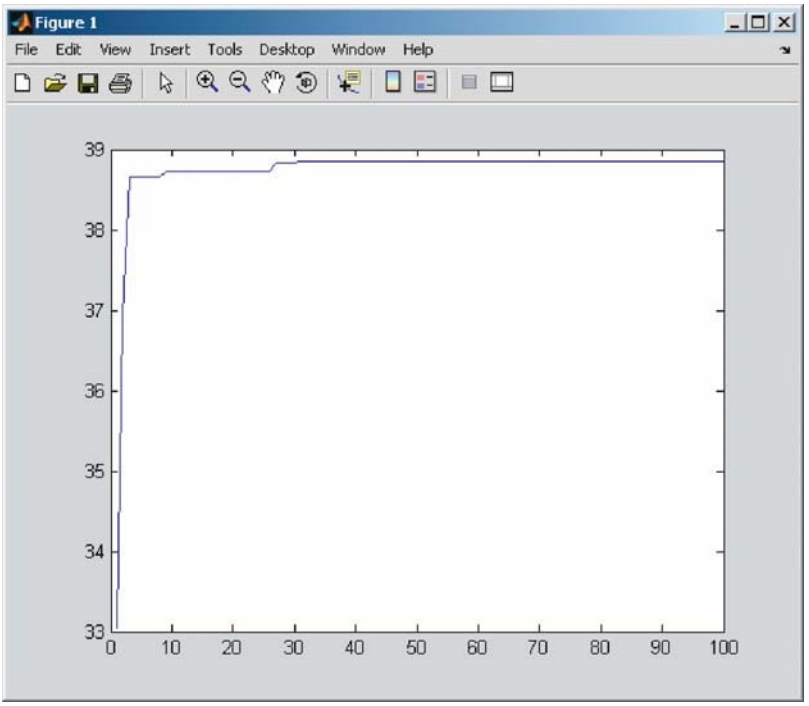


Fig. 3.23 Plot of best over time

```
11.6255  5.7250   38.8503
11.6255  5.8000   33.1253
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6255  5.7250   38.8503
```

The bestPop is the best solution tracked over generations and is given by

```
bestPop=
1.0000   12.0997  4.5040   34.1072
4.0000   11.1190  5.6165   37.4187
6.0000   12.0997  5.3230   38.2736
7.0000   12.1000  5.3230   38.2886
11.0000  12.1000  5.3230   38.2891
15.0000  12.0999  5.3237   38.3111
17.0000  12.0999  5.3241   38.3199
19.0000  12.1000  5.3241   38.3230
20.0000  12.0999  5.3249   38.3294
26.0000  12.1000  5.3249   38.3325
29.0000  12.1000  5.3252   38.3325
31.0000  12.1000  5.3250   38.3328
34.0000  12.0999  5.5258   38.5227
```

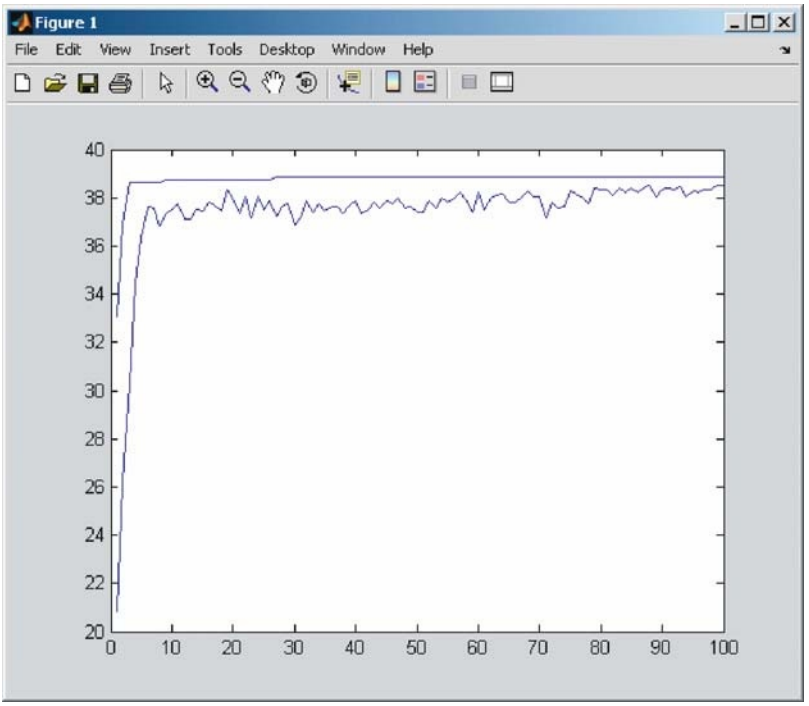


Fig. 3.24 Plot of the trace

```
39.0000  12.0999  5.5258   38.5227
41.0000  12.1000  5.5258   38.5241
42.0000  12.1000  5.5258   38.5258
49.0000  12.1000  5.5258   38.5259
50.0000  12.1000  5.5257   38.5263
52.0000  12.1000  5.5257   38.5277
53.0000  12.1000  5.5257   38.5285
58.0000  12.1000  5.5246   38.5311
73.0000  12.1000  5.5246   38.5311
77.0000  12.1000  5.5249   38.5327
102.0000 12.1000  5.7277   38.6529
121.0000 12.1000  5.7251   38.7327
125.0000 11.6235  5.7277   38.7663
127.0000 11.6235  5.7268   38.8115
128.0000 11.6235  5.7264   38.8260
138.0000 11.6242  5.7256   38.8449
143.0000 11.6243  5.7256   38.8455
149.0000 11.6255  5.7254   38.8486
154.0000 11.6258  5.7253   38.8492
156.0000 11.6255  5.7253   38.8494
161.0000 11.6258  5.7248   38.8498
```

```
165.0000 11.6258  5.7249   38.8501
167.0000 11.6257  5.7250   38.8503
174.0000 11.6257  5.7250   38.8503
176.0000 11.6256  5.7250   38.8503
178.0000 11.6256  5.7250   38.8503
182.0000 11.6256  5.7250   38.8503
200.0000 11.6255  5.7250   38.8503
```

The trace of the best value and average value of generations is trace and is given as

```
trace=
1.0000   34.1072  23.1787  5.0482
2.0000   34.1072  24.5508  4.9096
3.0000   34.1072  25.5157  4.7083
4.0000   37.4187  26.4310  4.4771
5.0000   37.4187  27.6779  5.4364
6.0000   38.2736  29.4413  6.9150
7.0000   38.2886  31.2012  6.2753
8.0000   38.2886  31.2902  7.6783
9.0000   38.2886  33.8514  7.1545
10.0000  38.2886  34.6541  6.4616
11.0000  38.2891  33.7751  7.4464
12.0000  38.2891  35.9043  3.8015
13.0000  38.2891  35.4520  4.3122
14.0000  38.2891  34.4458  5.8773
15.0000  38.3111  36.5369  3.4437
16.0000  38.3111  34.5778  5.5897
17.0000  38.3199  35.1337  5.0611
18.0000  38.3199  35.2378  4.3990
19.0000  38.3230  32.9923  7.8415
20.0000  38.3294  33.5600  7.8273
21.0000  38.3294  35.8216  4.0795
22.0000  38.3294  34.2110  5.8764
23.0000  38.3294  32.7150  8.0758
24.0000  38.3294  33.6589  5.5405
25.0000  38.3294  34.0214  4.6826
26.0000  38.3325  35.3198  5.1373
27.0000  38.3325  34.6960  3.9441
28.0000  38.3325  34.0835  4.7757
29.0000  38.3325  35.3622  3.7067
30.0000  38.3325  35.2971  4.1297
31.0000  38.3328  35.7535  4.5233
32.0000  38.3328  34.7642  5.3964
33.0000  38.3328  33.5016  6.0980
34.0000  38.5227  35.6655  4.2582
35.0000  38.5227  34.5819  5.2160
```

```
36.0000  38.5227  33.2062  5.9254
37.0000  38.5227  34.3777  5.6889
38.0000  38.5227  33.8986  6.2063
39.0000  38.5227  32.4974  6.8973
40.0000  38.5227  34.4664  5.8658
41.0000  38.5241  33.7668  5.5475
42.0000  38.5258  35.1506  4.0674
43.0000  38.5258  34.3509  5.8718
44.0000  38.5258  35.1174  3.4731
45.0000  38.5258  34.1712  5.2713
46.0000  38.5258  35.2121  4.5054
47.0000  38.5258  35.6970  3.2190
48.0000  38.5258  35.0480  5.6976
49.0000  38.5259  34.8284  5.0735
50.0000  38.5263  34.8945  6.1595
51.0000  38.5263  33.5793  6.3176
52.0000  38.5277  33.8005  7.5213
53.0000  38.5285  35.7316  4.7238
54.0000  38.5285  37.5123  2.0955
55.0000  38.5285  37.0287  2.9281
56.0000  38.5285  36.0784  4.5393
57.0000  38.5285  36.1575  3.9802
58.0000  38.5311  36.5405  3.6591
59.0000  38.5311  37.3714  2.9752
60.0000  38.5311  36.0265  5.1124
61.0000  38.5311  34.4864  5.5622
62.0000  38.5311  33.7935  5.7917
63.0000  38.5311  34.0268  5.7501
64.0000  38.5311  34.9396  5.2077
65.0000  38.5311  34.1589  5.2211
66.0000  38.5311  36.1598  3.7975
67.0000  38.5311  34.6644  5.6978
68.0000  38.5311  35.1792  4.8844
69.0000  38.5311  35.0514  4.6258
70.0000  38.5311  32.9727  7.5772
71.0000  38.5311  33.4051  6.6151
72.0000  38.5311  32.9887  8.4690
73.0000  38.5311  35.3436  5.3282
74.0000  38.5311  34.8642  5.1627
75.0000  38.5311  33.5429  6.7616
76.0000  38.5311  32.9961  6.2068
77.0000  38.5327  33.7914  4.6703
78.0000  38.5327  35.9305  4.3256
79.0000  38.5327  34.4854  4.8232
80.0000  38.5327  34.8661  5.8523
```

```
81.0000  38.5327  34.2859  4.9540
82.0000  38.5327  34.1750  4.3633
83.0000  38.5327  34.7919  4.7827
84.0000  38.5327  35.7043  4.1183
85.0000  38.5327  37.0428  3.0464
86.0000  38.5327  37.4018  2.1183
87.0000  38.5327  36.3151  3.9732
88.0000  38.5327  34.9434  6.7076
89.0000  38.5327  34.7943  4.1404
90.0000  38.5327  33.3296  4.7712
91.0000  38.5327  34.6687  4.1995
92.0000  38.5327  33.0360  7.9354
93.0000  38.5327  35.0913  4.8895
94.0000  38.5327  36.7091  3.6837
95.0000  38.5327  35.7297  4.7731
96.0000  38.5327  35.2076  5.4997
97.0000  38.5327  34.7146  6.8782
98.0000  38.5327  34.2732  7.2336
99.0000  38.5327  36.0443  5.3852
100.0000 38.5327  35.9575  4.2804
101.0000 38.5327  36.4166  3.6654
102.0000 38.6529  36.4502  3.6704
103.0000 38.6529  35.3717  4.3072
104.0000 38.6529  36.2800  4.7028
105.0000 38.6529  34.8762  4.8472
106.0000 38.6529  33.8093  5.5924
107.0000 38.6529  32.6878  5.9488
108.0000 38.6529  36.3054  5.0352
109.0000 38.6529  35.5506  4.9546
110.0000 38.6529  35.5196  4.9318
111.0000 38.6529  34.0884  6.7016
112.0000 38.6529  36.4098  4.8448
113.0000 38.6529  34.9755  6.6555
114.0000 38.6529  34.9621  5.3912
115.0000 38.6529  34.9344  4.8903
116.0000 38.6529  36.1622  4.0916
117.0000 38.6529  35.3084  4.5118
118.0000 38.6529  36.4299  4.0561
119.0000 38.6529  35.5365  4.6585
120.0000 38.6529  37.3656  2.8710
121.0000 38.7327  37.4827  3.0282
122.0000 38.7327  35.5222  6.8457
123.0000 38.7327  36.3370  4.5205
124.0000 38.7327  36.4651  4.2293
125.0000 38.7663  37.7130  2.6580
```

```
126.0000 38.7663  34.9901  5.1093
127.0000 38.8115  35.0062  5.4318
128.0000 38.8260  34.6968  6.5422
129.0000 38.8260  35.7437  4.6216
130.0000 38.8260  36.1102  4.1576
131.0000 38.8260  36.7162  4.8516
132.0000 38.8260  34.5489  4.5656
133.0000 38.8260  34.4837  5.0279
134.0000 38.8260  36.1054  4.3317
135.0000 38.8260  34.3368  6.9304
136.0000 38.8260  36.2440  4.5169
137.0000 38.8260  34.4543  7.0328
138.0000 38.8449  34.0382  7.6364
139.0000 38.8449  35.9950  5.3448
140.0000 38.8449  37.8867  1.7810
141.0000 38.8449  37.0922  2.9910
142.0000 38.8449  37.0558  2.3557
143.0000 38.8455  34.7951  6.4899
144.0000 38.8455  35.2430  4.8060
145.0000 38.8455  35.3846  5.6871
146.0000 38.8455  36.7674  4.5531
147.0000 38.8455  35.5666  4.8927
148.0000 38.8455  37.9294  2.9299
149.0000 38.8486  37.4416  3.4859
150.0000 38.8486  38.0338  1.9066
151.0000 38.8486  35.6859  5.5309
152.0000 38.8486  36.8254  4.1096
153.0000 38.8486  37.7455  2.5300
154.0000 38.8492  36.5524  3.5282
155.0000 38.8492  35.9343  4.1899
156.0000 38.8494  36.8985  3.4997
157.0000 38.8494  36.8765  3.9203
158.0000 38.8494  36.9165  4.1508
159.0000 38.8494  36.6652  2.8962
160.0000 38.8494  37.3308  2.5705
161.0000 38.8498  35.4290  3.9304
162.0000 38.8498  36.2275  3.9937
163.0000 38.8498  34.2875  4.8477
164.0000 38.8498  36.1114  3.9830
165.0000 38.8501  37.5243  2.7962
166.0000 38.8501  37.3780  3.7452
167.0000 38.8503  37.5425  3.2057
168.0000 38.8503  37.9579  2.8642
169.0000 38.8503  37.9634  2.7683
170.0000 38.8503  38.0747  2.3195
```

```
171.0000 38.8503  38.0051  2.1262
172.0000 38.8503  35.9455  4.5247
173.0000 38.8503  35.7065  4.3433
174.0000 38.8503  37.1020  2.8868
175.0000 38.8503  37.4196  2.8197
176.0000 38.8503  37.3590  2.7227
177.0000 38.8503  38.1987  1.9169
178.0000 38.8503  37.6946  2.3511
179.0000 38.8503  37.6201  2.2882
180.0000 38.8503  36.0645  5.1083
181.0000 38.8503  35.2126  5.9375
182.0000 38.8503  37.1521  4.1418
183.0000 38.8503  36.3624  4.6475
184.0000 38.8503  37.6765  2.8958
185.0000 38.8503  36.9854  3.3900
186.0000 38.8503  37.8195  2.1554
187.0000 38.8503  37.5148  2.9277
188.0000 38.8503  36.6619  3.3753
189.0000 38.8503  36.5639  3.5450
190.0000 38.8503  36.8821  2.5304
191.0000 38.8503  36.9445  2.9892
192.0000 38.8503  37.6581  4.1835
193.0000 38.8503  38.2271  1.7593
194.0000 38.8503  37.8886  2.3799
195.0000 38.8503  38.7010  0.4689
196.0000 38.8503  38.6092  0.7428
197.0000 38.8503  37.5890  2.8865
198.0000 38.8503  37.9686  1.9073
199.0000 38.8503  38.2437  1.5429
200.0000 38.8503  37.7615  0
```

Plot of the best over time:

After adding the average to the graph the plot is

The population size is increased by running the defaults and here x is the best solution found. x is given by

```
x=
11.6255  5.7250   38.8503
```

The end population is given as

```
endPop=
11.6255  5.7250   38.8503
11.6255  5.7250   38.8503
11.6255  5.7250   38.8503
11.6255  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
```

```
11.6255  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.5077   35.6719
11.6255  5.7250   38.8503
11.6255  5.7250   38.8503
11.6255  5.7250   38.8503
11.6255  5.7250   38.8503
11.6255  5.7250   38.8503
11.6255  5.7250   38.8503
11.6255  5.7250   38.8503
11.6255  5.7250   38.8503
11.6255  5.7250   38.8503
11.6255  5.3183   37.9774
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
12.1000  5.7250   38.7328
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.0815   28.4661
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.8000   33.1253
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
```

```
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
12.1000  5.7250   38.7328
11.6256  4.7863   29.4994
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  4.1000   33.1253
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
11.6256  5.7250   38.8503
```

The bestPop is the best solution tracked over generations and is given by

```
bestPop=
1.0000   11.6878  4.1351   33.0514
2.0000   11.6080  5.0151   36.9138
3.0000   12.1000  5.7277   38.6521
9.0000   12.1000  5.7248   38.7322
14.0000  12.1000  5.7248   38.7323
19.0000  12.1000  5.7252   38.7324
20.0000  12.1000  5.7250   38.7328
27.0000  11.6215  5.7250   38.8353
30.0000  11.6225  5.7250   38.8420
31.0000  11.6234  5.7251   38.8461
33.0000  11.6238  5.7251   38.8475
34.0000  11.6248  5.7253   38.8490
37.0000  11.6251  5.7253   38.8494
39.0000  11.6253  5.7253   38.8495
40.0000  11.6251  5.7249   38.8497
```

```
42.0000  11.6252  5.7252   38.8500
44.0000  11.6252  5.7251   38.8501
45.0000  11.6252  5.7251   38.8501
46.0000  11.6252  5.7251   38.8502
47.0000  11.6257  5.7251   38.8502
49.0000  11.6257  5.7251   38.8503
54.0000  11.6256  5.7251   38.8503
55.0000  11.6256  5.7250   38.8503
57.0000  11.6256  5.7250   38.8503
62.0000  11.6256  5.7250   38.8503
100.0000 11.6255  5.7250   38.8503
```

The trace of the best value and average value of generations is trace and is given as

```
trace=
1.0000   33.0514  20.8543  5.5915
2.0000   36.9138  26.1850  3.7594
3.0000   38.6521  30.2698  4.8529
4.0000   38.6521  34.4388  5.7638
5.0000   38.6521  36.3380  3.4856
6.0000   38.6521  37.6382  2.6525
7.0000   38.6521  37.5678  3.1902
8.0000   38.6521  36.8288  4.7221
9.0000   38.7322  37.3642  3.7018
10.0000  38.7322  37.5176  3.0986
11.0000  38.7322  37.7468  2.7063
12.0000  38.7322  37.1104  4.1060
13.0000  38.7322  37.1293  4.5199
14.0000  38.7323  37.5206  3.4273
15.0000  38.7323  37.4392  3.4050
16.0000  38.7323  37.8054  2.2823
17.0000  38.7323  37.6640  3.0310
18.0000  38.7323  37.4535  3.3716
19.0000  38.7324  38.3186  1.4077
20.0000  38.7328  37.9625  2.3089
21.0000  38.7328  37.3650  3.1883
22.0000  38.7328  38.0327  2.4146
23.0000  38.7328  37.1694  4.3284
24.0000  38.7328  38.0157  2.2604
25.0000  38.7328  37.5252  4.0997
26.0000  38.7328  37.8736  2.6568
27.0000  38.8353  37.2192  4.5541
28.0000  38.8353  37.6421  3.4835
29.0000  38.8353  37.7363  2.8391
30.0000  38.8420  36.8445  5.3177
31.0000  38.8461  37.1877  4.4286
```

```
32.0000  38.8461  37.8942  2.9799
33.0000  38.8475  37.3862  3.6901
34.0000  38.8490  37.7391  2.9670
35.0000  38.8490  37.4372  4.2490
36.0000  38.8490  37.6387  3.4197
37.0000  38.8494  37.6098  3.4092
38.0000  38.8494  37.3225  3.9644
39.0000  38.8495  37.6767  3.5328
40.0000  38.8497  37.8728  2.7474
41.0000  38.8497  37.3336  3.9800
42.0000  38.8500  37.4863  3.8793
43.0000  38.8500  37.7863  2.7172
44.0000  38.8501  37.5445  3.6816
45.0000  38.8501  37.8737  3.2160
46.0000  38.8502  37.7545  2.9037
47.0000  38.8502  37.9937  2.4315
48.0000  38.8502  37.5753  3.2589
49.0000  38.8503  37.6196  3.2681
50.0000  38.8503  37.4063  4.3144
51.0000  38.8503  37.4233  3.6007
52.0000  38.8503  37.8686  2.9723
53.0000  38.8503  37.5677  3.5648
54.0000  38.8503  37.9740  2.8806
55.0000  38.8503  37.8308  2.7412
56.0000  38.8503  37.9925  2.6881
57.0000  38.8503  38.2237  1.8520
58.0000  38.8503  37.9274  2.4938
59.0000  38.8503  37.3781  4.2441
60.0000  38.8503  38.2147  2.4868
61.0000  38.8503  37.5436  3.7695
62.0000  38.8503  38.0116  2.6923
63.0000  38.8503  38.1042  2.3429
64.0000  38.8503  38.1858  2.2369
65.0000  38.8503  37.7888  2.9087
66.0000  38.8503  37.8192  3.3599
67.0000  38.8503  37.9757  2.5590
68.0000  38.8503  38.2599  2.1431
69.0000  38.8503  38.0134  2.6469
70.0000  38.8503  38.0298  2.9759
71.0000  38.8503  37.1542  4.9477
72.0000  38.8503  37.7780  3.3586
73.0000  38.8503  37.5847  3.6610
74.0000  38.8503  37.6548  3.6823
75.0000  38.8503  38.2937  2.5445
76.0000  38.8503  38.1628  2.2277
```

```
77.0000  38.8503  37.9860  2.6177
78.0000  38.8503  37.7496  3.1160
79.0000  38.8503  38.4028  1.6012
80.0000  38.8503  38.3281  1.6424
81.0000  38.8503  38.3394  1.7726
82.0000  38.8503  38.0732  2.9762
83.0000  38.8503  38.3923  1.8357
84.0000  38.8503  38.2261  2.3682
85.0000  38.8503  38.4105  1.8514
86.0000  38.8503  38.1992  2.4550
87.0000  38.8503  38.4027  1.7357
88.0000  38.8503  38.4973  1.2981
89.0000  38.8503  38.0599  2.9152
90.0000  38.8503  38.3130  2.0107
91.0000  38.8503  38.4035  1.9600
92.0000  38.8503  38.3579  2.0093
93.0000  38.8503  38.4365  1.7929
94.0000  38.8503  38.0592  2.6565
95.0000  38.8503  38.3033  1.9880
96.0000  38.8503  38.2454  1.9513
97.0000  38.8503  38.3060  2.2036
98.0000  38.8503  38.3576  2.0398
99.0000  38.8503  38.5135  1.7032
100.0000 38.8503  38.5367  0
```

Plot of the best over time:

After adding the average to the graph the plot is

Thus we conclude that, while increasing the population size, better results are obtained.

3.16.6 Illustration 5 – Constrained Problem

Constrained problem - To draw the biggest possible circle in a 2D space filled with stars without enclosing any of them.

Solution:

The parameters are initialized and the stars are built. The initial population is created and the fitness is assigned. The iterations are performed until the best values are obtained.

MATLAB Code:

Initializing parameters

```
clear;
stars=30;
iterations=500;
population=100;
mutation=0.7;
```

```
mutation_step=0.8;
grid_size=10;
```

Building the stars

```
for i=1:stars
for j=1:2
star(i,j)=rand(1)*grid_size;
end
end
```

Creating initial population

```
for i=1:population
for j=1:2
cit(i,j)=rand(1)*grid_size;
end
end
```

Initial Population Fitness Assignment

```
for j=1:population
for k=1:stars
d(k)=sqrt((star(k,1)-cit(j,1))^2+(star(k,2)-cit(j,2))
^2);
end
d(stars+1)=cit(j,1);
d(stars+2)=grid_size - cit(j,1);
d(stars+3)=cit(j,2);
d(stars+4)=grid_size - cit(j,2);
d=sort(d);
cit(j,3)=d(1);
end
cit=sortrows(cit,-3);
cit(1,:)
cit(:,3)=cit(:,3)/cit(1,3);
```

The Iterations are performed and the stars are plotted (Figure 3.25) with the largest circle without enclosing any of the stars.

```
for i=1:iterations
%--------------- Mating Selection -----------------
cit(:,3)=cit(:,3)/cit(1,3);
pool=[];
for l=1:population
if rand(1)<cit(l,3),pool=[pool;cit(l,:)];,end
end
%--------------- Crossover ------------------
s=size(pool,1);
if s/2 - round(s/2) ~=0, pool=pool(1:s-1,:);, s=size
(pool,1);, end
for m=1:2:s
```

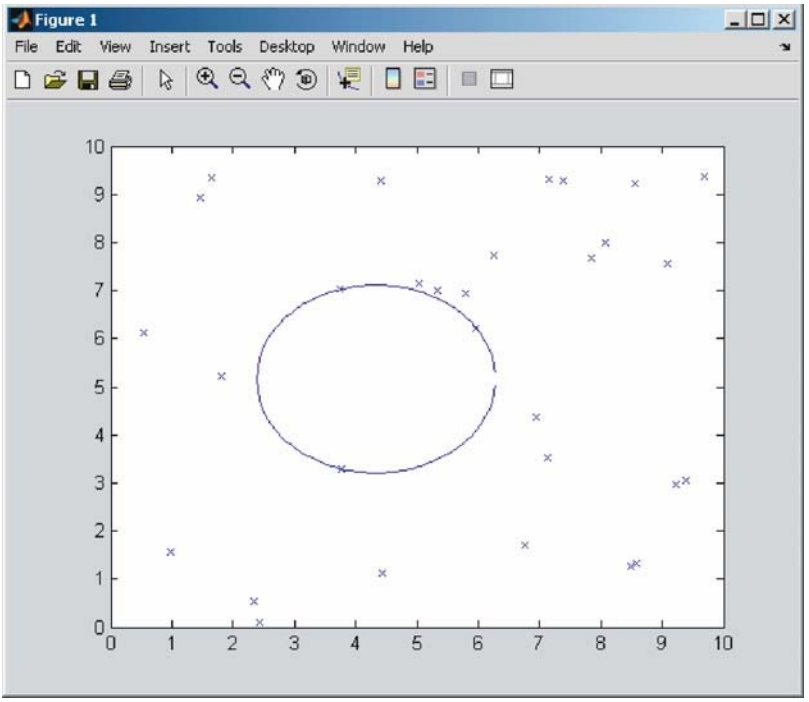


Fig. 3.25 Plot of the biggest possible circle without enclosing any of the stars

```
pool=[pool ; [pool(m,1), pool(m+1,2), 0]];
pool=[pool ; [pool(m+1,1), pool(m,2), 0]];
end
%--------------- Mutation -------------------
s=size(pool,1);
for m=1:s
if ((rand < mutation) & (pool(m,1) < grid_size -
mutation_step)), pool(m,1)=pool(m,1)+(2*rand - 1)
*mutation_step;,end
if ((rand < mutation) & (pool(m,2) < grid_size -
mutation_step)), pool(m,2)=pool(m,2)+(2*rand - 1)
*mutation_step;,end
end
%--------------- Environmental Selection------------
temp=[cit ; pool];
ts=size(temp,1);
for j=1:ts
for k=1:stars
d(k)=sqrt((star(k,1)-temp(j,1))^2+(star(k,2)-temp(j,2))^2);
end
```

```
d(stars+1)=temp(j,1);
d(stars+2)=grid_size - temp(j,1);
d(stars+3)=temp(j,2);
d(stars+4)=grid_size - temp(j,2);
d=sort(d);
temp(j,3)=d(1);
end
temp=sortrows(temp,-3);
cit=temp(1:population,:);
end
xc=cit(1,1);
yc=cit(1,2);
r=cit(1,3);
x=(xc-r) : 0.05 : (xc+r);
for i=1:size(x,2)
y(i)=yc+sqrt(r^2-(x(i)-xc)^2);
end
plot (x,y)
for i=1:size(x,2)
y(i)=yc - sqrt(r^2-(x(i)-xc)^2);
end
plot (x,y)
hold off
```

Output:

Points of the stars which form the largest circle are

```
[4.96148667045705 4.80247420429890 1.74246535685782]
```

3.16.7 Illustration 6 – Maximum of any given Function

To find the maximum of any given function. The function can be introduced in fun.m file in current directory.

Solution:

The maximum of any function can be determined by introducing that function in fun.m

MATLAB Code:

The fun.m routine

```
function z=fun(x)
z=0;
for i=1:length(x)
z=z+x(i)*sin(abs(x(i)));
```

```
end
```

After specifying the required function to be maximized in fun.m file, the optimizing routine starts

```
function GAconstrain
tic
clc
figure(1)
clf
clear all
format long
As an initial process the parameters are set
var=2; % Number of variables (this item must be equal
       % to the
       % number of variables that is used in the
         function in
       % fun.m file)
n=100; % Number of population
m0=30; % Number of generations that max value remains
% constant
       % (use for termination criteria)
nmutationG=20; %number of mutation children(Gaussian)
nmutationR=20; %number of mutation children(Random)
nelit=2; %number of elitism children
valuemin=ones(1,var)*-5*pi; % min possible value
 of variables
valuemax=ones(1,var)*5*pi; % max possible value of
 variables
A=[1 1;-0.5+1;1 -1;-1 -1]; % constrains: Ax+B>0
B=[5*pi 0 5*pi+5*pi+10];
[r,c]=size(A);
if c~=var
disp('Number of columns in A must be equivalent to
 var')
   A
   var
   return
end
nmutation=nmutationG+nmutationR;
sigma=(valuemax-valuemin)/10; % Parameter that related
                              % to Gaussian
                              % function and used in
                              % mutation step
max1=zeros(nelit,var);
parent=zeros(n,var);
cu=[valuemin(1) valuemax(1) valuemin(2) valuemax(2)];
for l=1:var
```

```
p(:,l)=valuemin(l)+rand(n,1).*(valuemax(l)-valuemin
(l));
end
initial=p;
m=m0;
maxvalue=ones(m,1)*-1e10;
maxvalue(m)=-1e5;
g=0;
meanvalue(m)=0;
```

Termination criteria is given as

```
while abs(maxvalue(m)-maxvalue(m-(m0-1)))>0.001
  *maxvalue(m) & ...
(abs(maxvalue(m))>1e-10 & abs(maxvalue(m-(m0-1)))
>1e-10)...
& m<10000 & abs(maxvalue(m)-meanvalue(m))>1e-5 | m<20
sigma=sigma./(1.05);% reducing the sigma value
% ------ **** % reducing the number of mutation()
random **** ----
g=g+1;
if g>10 & nmutationR>0
g=0;
nmutationR=nmutationR-1;
nmutation=nmutationG+nmutationR;
end
```

Function evaluation

```
for i=1:n
y(i)=fun00(p(i,:));
flag(i)=0;
end
if var==2
figure(1)
hold off
plot00(cu)
hold on
plot3(p(:,1),p(:,2),y,'ro')
title({'Genetic Algorithm: constrained problem'...
,'Performance of GA (o : each individual)',...
'blue o:possible solution & red o impossible
 solution',...
'black o: best solution'},'color','b')
end
s=sort(y);
maxy=max(y);
miny=min(y);
for i=1:n
```

```
k=length(B);
for j=1:k
x=p(i,:);
if (A(j,:)*x'+B(j))<=0
y(i)=miny;
flag(i)=1;
end
end
end
if var==2
for i=1:n
if flag(i)==0
plot3(p(i,1),p(i,2),y(i),'bo')
end
end
end
s=sort(y);
maxvalue1(1:nelit)=s(n:-1:n-nelit+1);
if nelit==0
maxvalue1(1)=s(n);
for i=1:n
if y(i)==maxvalue1(1)
max1(1,:)=p(i,:);
end
end
end
for k=1:nelit
for i=1:n
if y(i)==maxvalue1(k)
max1(k,:)=p(i,:);
end
end
end
if var==2
hold on
plot3(max1(1,1),max1(1,2),maxvalue1(1),'kh')
end
y=y-min(y)*1.02;
sumd=y./sum(y);
meanvalue=y./(sum(y)/n);
```

Roulette wheel selection process is performed

```
for l=1:n
sel=rand;
sumds=0;
j=1;
```

```
while sumds<sel
sumds=sumds+sumd(j);
j=j+1;
end
parent(l,:)=p(j-1,:);
end
p=zeros(n,var);
```

Regeneration

```
for l=1:var
```

Crossover

```
for j=1:ceil((n-nmutation-nelit)/2)
t=rand*1.5-0.25;
p(2*j-1,l)=t*parent(2*j-1,l)+(1-t)*parent(2*j,l);
p(2*j,l)=t*parent(2*j,l)+(1-t)*parent(2*j-1,l);
end
```

Elitism

```
for k=1:nelit
p((n-nmutation-k+1),l)=max1(k,l);
end
```

Mutation

```
for i=n-nmutation+1:n-nmutationR
phi=1-2*rand;
z=erfinv(phi)*(2^0.5);
p(i,l)=z*sigma(l)+parent(i,l);
end
for i=n-nmutationR+1:n
p(i,1:var)=valuemin(1:var)+rand(1,var).*(valuemax
(1:var)...
-valuemin(1:var));
end
for i=1:n
for l=1:var
if p(i,l)<valuemin(l)
p(i,l)=valuemin(l);
elseif p(i,l)>valuemax(l)
p(i,l)=valuemax(l);
end
end
end
end
p;
m=m+1;
max1;
maxvalue(m)=maxvalue1(1);
```

```
maxvalue00(m-m0)=maxvalue1(1);
mean00(m-m0)=sum(s)/n;
meanvalue(m)=mean00(m-m0);
figure(1)
if var~=2
hold off
plot(maxvalue00)
hold on
plot(mean00,'g')
hold on
title({'Performance of GA',...
'best value GA:blue, mean value GA:green',''}...
,'color','b')
xlabel('number of generations')
ylabel('value')
end
pause(0.001)
end
```

Plot of the performance of GA (Figure 3.26)

```
clc
num_of_fun_evaluation=n*m
max_point_GA=max1(1,:)
maxvalue_GA=maxvalue00(m-m0)
if var==2
figure(1)
hold on
plot3(max1(1,1),max1(1,2),maxvalue1,'yp')
hold on
end
```

Plot of the best value (Figure 3.27)

```
figure(2)
title('Performance of GA(best value)','color','b')
xlabel('number of generations')
ylabel('max value of best solution')
hold on
plot(maxvalue00)
hold on
toc
function z=fun00(x)
z=0;
for i=1:length(x)
z=z+x(i)*sin(abs(x(i)));
end
function plot00(cx)
```

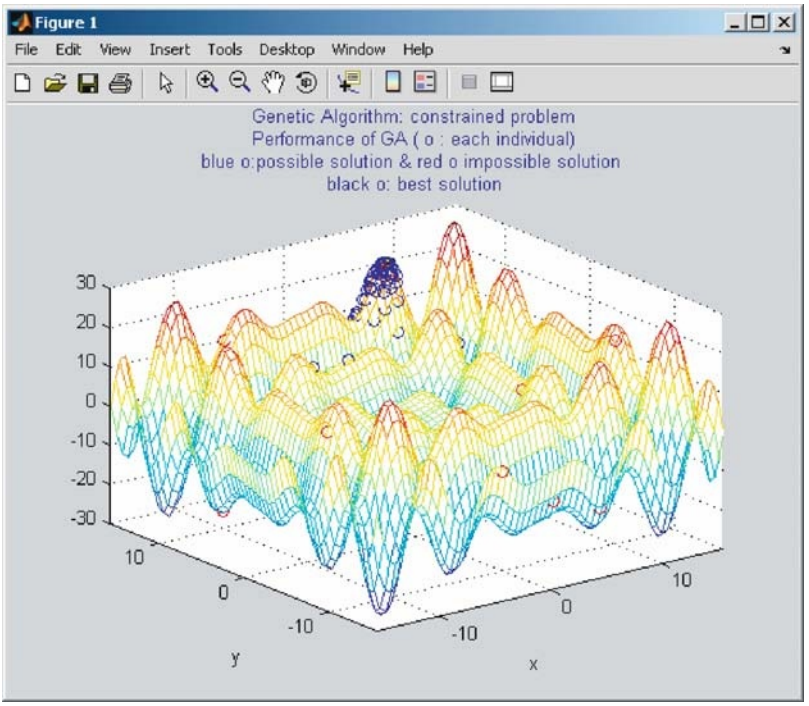


Fig. 3.26 Performance of GA

```
ezmesh('x*sin(abs(x))+y*sin(abs(y))',cx)
```

Output:

```
num_of_fun_evaluation - 7400
max_point_GA          - [7.97257748749159
                        14.22297714288591]
maxvalue_GA           - 22.08722071228645
Elapsed Time          - 16.092657 seconds
```

Thus it is clear that any function that can be maximized can be inserted in the fun.m file and the performance can be obtained.

Summary

Thus it is observed that genetic algorithms are robust, useful, and are most powerful apparatuses in detecting problems in an array of fields. In addition, genetic algorithms unravel and resolve an assortment of complex problems. Moreover, they are capable of providing motivation for their design and foresee broad propensity of the innate systems. Further, the reason for these ideal representations is to provide thoughts on the exact problem at hand and to examine their plausibility. Hence,

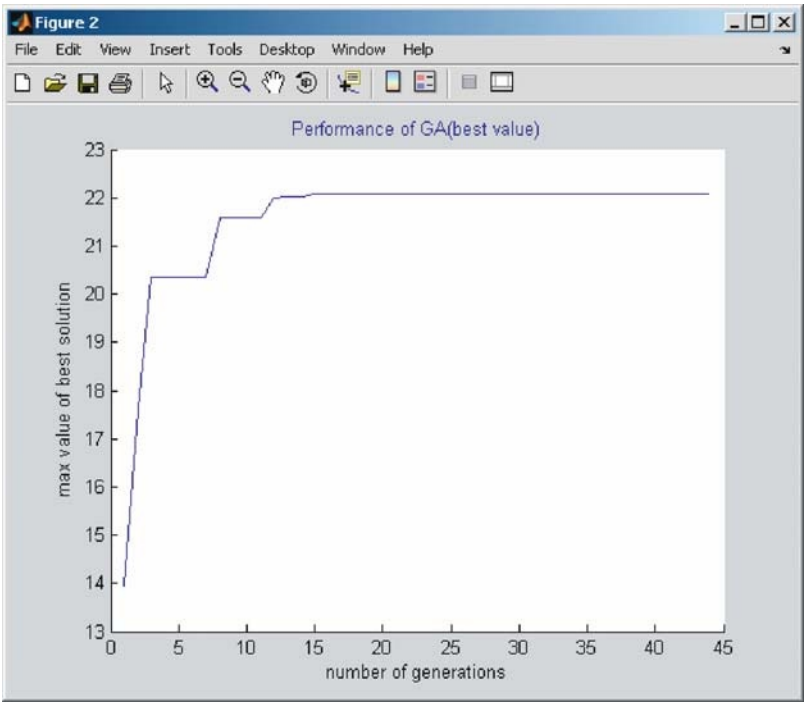


Fig. 3.27 Plot of the best value

employing them as a computer encode and identify how the propensities are affected from the transformations with regards to the model. Without genetic algorithms, it is not possible to solve real world issues. The genetic algorithm methods may permit researchers to carry out research, which was not perceivable during this evolutionary technological era. Therefore, one is able to replicate this phenomenon, which would have been virtually impossible to obtain or analyze through traditional methods or through the analysis of certain equations. Because GAs form a subset field of evolutionary computation, optimization algorithms are inspired by biological and evolutionary systems and provide an approach to learning that is based on simulated evolution. Given a basic understanding of biological evolution and Darwin's ideas of the survival of the fittest in Chapter 1, one is able to comprehend the computational applications of such a theory wherein a computer analyzes populations of data and "learns" which data strings to follow and repeat based on the most "fit" or successful data in a population. This chapter has provided the basics of GA with illustrative examples. A set of MATLAB illustrations were also included to provide a practical knowledge to the user. Genetic algorithms have proven time and again that they are excellent means to solve real world and complex problems.

Review Questions

1. What do you mean by Calculus-based schemes and Enumerative-based optimization schemes?
2. Define Genetic Algorithm.
3. Explain the functionality of GA with a suitable flow chart.
4. Mention some of the common genetic representations.
5. State Schema Theorem.
6. Derive schema theorem using the mathematical model.
7. How are the destructive effects of schema theorem compensated?. Explain in terms of crossover and mutation.
8. Mention the considerations while constructing a fitness function.
9. Explain inversion and reordering.
10. Write a note on Epistatis.
11. Define deception.
12. What do you mean by naive evolution?
13. Define the process speciation.
14. Explain briefly on restricted mating, diploidy and dominance.
15. What are the important issues that are considered while implementing a GA?
16. Compare GA with Neural Nets, Random Search and Simulated Annealing.
17. Mention the types of GA.
18. Explain the sequential GA with a suitable algorithm.
19. Explain the parallel GA with a suitable algorithm.
20. Briefly explain the steps of operation in a Hybrid GA.
21. Describe the design of crossover and mutation probabilities in Adaptive GA.
22. Explain the functionality of a fast Messy GA with the aid of a suitable flowchart.
23. Explain the algorithm of SSGA and GGA.
24. Mention the advantages of GA.
25. Mention a few application areas of GA.

Chapter 4
Genetic Programming Concepts

Learning Objectives: On completion of this chapter the reader will have knowledge on:

- Basic definition of Genetic Programming
- History of Genetic Programming
- Basics of Lisp Programming Language
- Creating an individual using trees in GP
- Creating a Random Population
- Performing the Fitness Test
- Definitions of Functions and Terminals
- The Genetic Operations such as Selection, Crossover, Mutation and how they differ from Genetic Algorithms
- The Preparatory Steps of Genetic Programming
- Flow Chart of GP
- Variants of GP such as Meta GP, Cartesian GP and Strongly Typed GP

4.1 Introduction

Genetic programming is a systematic method for getting computers to automatically solve a problem starting from a high-level statement of what needs to be done. Genetic Programming(GP), one of a number of evolutionary algorithms, follows Darwin's theory of evolution—often paraphrased as "survival of the fittest". There is a population of computer programs (individuals) that reproduce with each other. Over time, the best individuals will survive and eventually evolve to do well in the given environment.

GP definition - "*Genetic programming is an automated method for creating a working computer program from a high-level problem statement of a problem. Genetic programming does this by genetically breeding a population of computer programs using the principles of Darwinian natural selection and biologically inspired operations*."

The idea of a computer automatically programming itself is a very old, desirable and elusive goal. Automatic Programming has, in the past, had a very bad reputation. This is probably because, intuitively, it would seem that developing software which is able to write software should be easier that writing software able to prove theorems, or paint pictures, etc. However, computer programming is a very difficult task, which involves intelligence, creativity, understanding, cunning and guile. In short, it is as difficult to get a computer to program as it is to get it to do anything else. So, when early attempts at automatic programming largely failed to deliver what they promised, people began to avoid using the term automatic programming, and people generally stayed away from the subject.

Many Artificial Intelligence (AI) techniques disguise the fact that, to some extent, they are performing automated programming. If decision tree learning techniques are considered, for example, the end product is a decision tree, which if it is to be used to actually make decisions for us, has to be "executed", i.e., information about the system is going to be given as input, and an answer will be computed. In this case, the program doing the computing is fairly simple - decision trees can be easily translated into a bunch of "if-then" statements. It's a similar situation with Artificial Neural Networks. So, many AI techniques are kind-of doing automated programming. The Genetic Programming (GP) developers are more explicit about this - they state clearly that their GP engines program software automatically.

The way Genetic Programming engines generate programs is by following an evolutionary approach. The general approach is as follows. The user specifies the task to be undertaken (or problem to be solved) using an evaluation function to express what the evolved programs should do. They also specify what kind of things the programs will be able to use during the computation, e.g., whether or not they will be able to multiply two numbers together. Then, an initial population of programs is generated at random. Each program is translated, compiled and executed and how well it performs with respect to the task is assessed. This enables the calculation of a fitness value for each of the programs, and the best of them are chosen for reproduction. Programs are combined or mutated into offspring, which are added to the next generation of programs. This process repeats until a termination condition is met. In this chapter, a brief history of genetic programming is discussed. To get an idea about programming a basic introduction to Lisp Programming Language is dealt. The basic operations of GP are discussed along with an illustration. The steps of GP are also illustrated along with a flow chart and enhanced versions of GP such as Meta GP, Cartesian GP and Strongly Typed GP are also elaborated in this chapter.

In Genetic Programming programs are evolved to solve pre-described problems from both of these domains. The term evolution refers to an artificial process gleaned from natural evolution of living organisms. This process has been abstracted and stripped off of most of its intricate details. It has been transferred to the world of algorithms where it can serve the purpose of approximating solutions to given or even changing problems (machine learning) or for inducing precise solutions in the form of grammatically correct (language) structures (automatic programming).

It has been realized that the representation of programs, or generally structures, has a strong influence on the behavior and efficiency of the resulting algorithm.

As a consequence, many different approaches toward choosing representations have been adopted in Genetic Programming. The principles have been applied even to other problem domains such as design of electronic circuits or art and musical composition.

Genetic Programming is also part of the growing set of Evolutionary Algorithms which apply the search principles of natural evolution in a variety of different problem domains, notably parameter optimization. Evolutionary Algorithms, and Genetic Programming in particular, follow Darwin's principle of differential natural selection. This principle states that the following preconditions must be fulfilled for evolution to occur via (natural) selection

- There are entities called individuals which form a population. These entities can reproduce or can be reproduced.
- There is heredity in reproduction, that is to say that individuals produce similar offspring.
- In the course of reproduction there is variety which affects the likelihood of survival and therefore of reproducibility of individuals.
- There are finite resources which cause the individuals to compete. Due to over reproduction of individuals not all can survive the struggle for existence. Differential natural selections will exert a continuous pressure towards improved individuals.

In the long run, genetic programming and its kin will revolutionize program development. Present methods are not mature enough for deployment as automatic programming systems. Nevertheless, GP has already made inroads into automatic programming and will continue to do so in the foreseeable future. Likewise, the application of evolution in machine-learning problems is one of the potentials that is to be exploited over the coming decade.

GP is part of a more general field known as evolutionary computation. Evolutionary computation is based on the idea that basic concepts of biological reproduction and evolution can serve as a metaphor on which computer-based, goal-directed problem solving can be based. The general idea is that a computer program can maintain a population of artifacts represented using some suitable computer-based data structures. Elements of that population can then mate, mutate, or otherwise reproduce and evolve, directed by a fitness measure that assesses the quality of the population with respect to the goal of the task at hand.

For example, an element of a population might correspond to an arbitrary placement of eight queens on a chessboard, and the fitness function might count the number of queens that are not attacked by any other queens. Given an appropriate set of genetic operators by which an initial population of queen placements can spawn new collections of queen placements, a suitably designed system could solve the classic eight-queens problem. GP's uniqueness comes from the fact that it manipulates populations of structured programs—in contrast to much of the work in evolutionary computation in which population elements are represented using flat strings over some alphabet.

The central challenge and common goal of AI and machine learning is to get a computer to solve a problem without explicitly programming it. This challenge

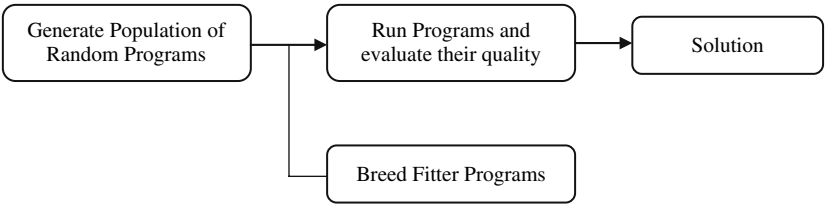


Fig. 4.1 Main loop of genetic programming

envisions an automatic system whose input is a high level statement of a problem's requirements and whose output is a satisfactory solution to the given problem. To be useful, the system must routinely achieve this goal at levels that equal or exceed the human level of performance.

Genetic programming is a domain-independent method that genetically breeds a population of computer programs to solve a problem. Specifically, genetic programming iteratively transforms a population of computer programs into a new generation of programs by applying analogs of naturally occurring genetic operations. This process is illustrated in Figure 4.1.

The genetic operations include crossover (sexual recombination), mutation, reproduction, gene duplication, and gene deletion. Analogs of developmental processes are sometimes used to transform an embryo into a fully developed structure. Genetic programming is an extension of the genetic algorithm in which the *structures* in the population are not fixed-length character strings that encode candidate solutions to a problem, but *programs* that, when executed, *are* the candidate solutions to the problem.

The most commonly used representation used in genetic programming is the program tree (or parse tree or syntax tree). In fact, this representation is strongly linked with the term "genetic programming", and many might use a different term to refer to the evolution of programs where the programs have a different representation. Some alternative program representations include finite automata (evolutionary programming) and grammars (grammatical evolution). For example, the simple expression $\max(x^*x,\ x+3^*y)$is represented as shown in Figure 4.2. The tree includes *nodes* (which is also called *point*) and *links*. The nodes indicate the instructions to

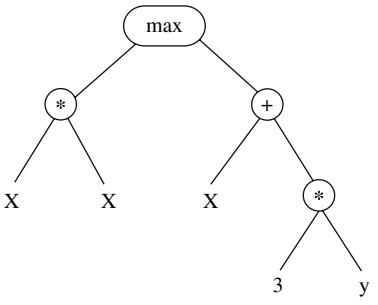


Fig. 4.2 Basic tree-like program representation used in genetic programming

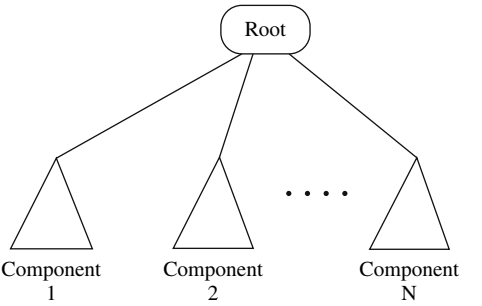


Fig. 4.3 Multi-tree program representation

execute. The links indicate the arguments for each instruction. In the following the internal nodes in a tree will be called *functions*, while the tree's leaves will be called *terminals*.

In more advanced forms of genetic programming, programs can be composed of multiple components (e.g., subroutines). In this case the representation used in genetic programming is a set of trees (one for each component) grouped together under a special node called *root*, as illustrated in Figure 4.3. These subtrees are called *branches*. The number and type of the branches in a program, together with certain other features of the structure of the branches, form the *architecture* of the program. Genetic programming trees and their corresponding expressions can equivalently be represented in *prefix notation* (e.g., as Lisp S-expressions). Knowledge about the representation of Lisp programs is helpful for understanding of genetic programming. A short description on Lisp programming is discussed in section 4.3.

In prefix notation, functions always precede their arguments. For example, $\max(x^*x,\ x+3^*y)$ becomes $(\max\ (^*\ x\ x)\ (+x(^*3\ y)))$. In this notation, it is easy to see the correspondence between expressions and their syntax trees. Simple recursive procedures can convert prefix-notation expressions into infix-notation expressions and vice versa.

4.2 A Brief History of Genetic Programming

In 1992 John Koza proposed a new significantly new approach - Genetic Programming (GP). In GP individual population members (chromosomes) are not fixed length linear character strings that encode possible solution to the problem like in GA, but they are programs that, when executed, provide solution to the problem. These programs are expressed in GP as parse trees of varying sizes and shapes, what makes these methods flexible in their application to the wide range of problems. The difference in chromosomes representation is the main and almost the only difference between this method and GA. The overall Darwinian idea of survival stays the same, but there are changes in mutation and crossover operators and in fitness function calculation. Instead of mutation of an individual gene in a string chromosome, in GP mutation operator may regenerate as a single tree node, as a whole

sub tree of the chromosome's tree. The same happens with crossover – instead of exchanging with chromosome parts of the same length, in GP chromosomes exchange with their sub trees, which may differ as in size, as in shape. However it is always required to do some checking of chromosomes and trim them in case they are growing too long. To compute fitness value of individual chromosome in GP, the chromosome is not just passed as value to some sort of function, which computes the fitness value. Instead of this, the program, which is represented by the chromosome, is executed and fitness value is calculated depending on the output of the program.

4.3 The Lisp Programming Language

Lisp was invented by John McCarthy and others starting in 1958. This is the second oldest programming language which is still in common use today. It was completely different from the other languages of the day (FORTRAN and COBOL). Pure Lisp has only two kinds of data structures, the ***a***tom and the ***l***ist. Atoms are either symbols or numbers. Lists are linked lists where the elements of the list are either atoms or other lists. Lists of atoms are written as follows:

```
(A B 4 5)
```

Nested list structures (lists with lists as elements) are written as follows:

```
(A (B C) D (E (F G)))
```

This example is a list of four elements. The first is the atom `A`; the second is the sublist `(B C)`; the third is the atom `5`, and the fourth is the sublist `(E (F G))`. Internally, lists are usually represented as single-linked lists. Each node of the list consists of two pointers. The first pointer points either to the corresponding atom or to the corresponding sublist. The second pointer points to the next node in the list. Pictures will be shown in class. Note that a nested list structure is just a representation of a tree.

Pure Lisp is a functional programming language. Computation in a functional program is accomplished by applying functions to arguments. These arguments may be symbols, numbers, or other function calls. Neither assignment statements nor variables are needed. Iterative processes can be specified with recursion. Thus, a Lisp program is just a collection of functions that can call each other. Functions in Lisp are represented in the same way as data, namely as nested list structures. Here is a function that squares its argument `x`:

```
(defun square (x)(* x x))
```

The `defun` can be thought of as a key word that indicates that a function definition follows. The name of the function is `square`. Then follows a list of arguments (dummy parameters). Finally is the body of the function. The function returns the value resulting from multiplying `x` times `x`.

Here is a slightly more complex example that uses the function `square`:

```
(defun hypotenuse(a b))
     (sqrt (+(square a) (square b))))
```

The function to find the length is defined as

```
(defun list-length(x)
(if (null x)
       0
       (+1 (length(rest x)))))
```

If the list is `null`(i.e., empty), then the length is `0`. Otherwise, the length is `1` plus the length of the list with the first element removed. Since Lisp functions are stored as data, it is very easy to write functions that define and manipulate other functions. This is exactly what genetic programming does. It is relatively easy to write a simple genetic programming package in Lisp.

4.4 Operations of Genetic Programming

The operational steps of a typical genetic programming is described in this section.

4.4.1 Creating an Individual

The fundamental elements of an individual are its genes, which come together to form code. An individual's program is a tree-like structure and as such there are two types of genes: *functions* and *terminals*.

Terminals, in tree terminology, are leaves (nodes without branches) while functions are nodes with children. The function's children provide the arguments for the function. In the example (Figure 4.4) there are two functions ($+$ and $\times$) and three terminals (3, y and 6). The $\times$ (multiplication) function requires two arguments: the 3 and the return value of the subtree whose root is $+$. The $+$ (addition) function also takes two arguments which are provided by the x and the 6.

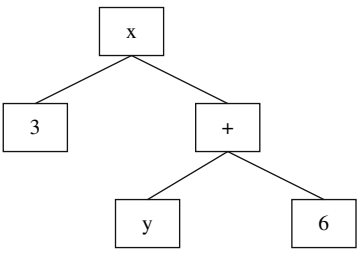


Fig. 4.4 A typical simple individual that returns 3(y+6)

The example may be interpreted as $3 \times (y + 6)$. The genes that will be available to the GP system must be selected or created by the user. This is an important decision as poor selection may well render the system incapable of evolving a solution.

4.4.2 Creating a Random Population

Assuming the genes have been chosen, the first step is to create the random population. Three techniques specified to create random population namely *grow, full* and *ramped-half-and-half* are *discussed below*:

Grow

With this first technique the entire population is created by using the grow method which creates one individual at a time. An individual created with this method may be a tree of any depth up to a specified maximum, m.

1. Starting from the root of the tree every node is randomly chosen as either a function or terminal.
2. If the node is a terminal, a random terminal is chosen.
3. If the node is a function, a random function is chosen, and that node is given a number of children equal to the arity (number of arguments) of the function. For every one of the function's children the algorithm starts again, unless the child is at depth m, in which case the child is made a randomly selected terminal. This method does not guarantee individuals of a certain depth (although they will be no deeper than m). Instead it provides a range of structures throughout the population The method may produce individuals containing only one (terminal) node. Such individuals are quickly bred out if the problem is non-trivial, and so are not really much value.

Full

The full method is very similar to the grow method except the terminals are guaranteed to be a certain depth. This guarantee does not specify the number of nodes in an individual. This method requires a final depth, d.

1. Every node, starting from the root, with a depth less than d, is made a randomly selected function. If the node has a depth equal to d, the node is made a randomly selected terminal.
2. All functions have a number (equal to the arity of the function) of child nodes appended, and the algorithm starts again. Thus, only if d is specified as one, could this method produce a one-node tree.

Ramped-half-and-half

To increase the variation in structure both grow and full methods can be used in creating the population—this technique, ramped-half-and-half, is the sole method used by Koza. Only a maximum depth, *md*, is specified but the method generates a population with a good range of randomly sized and randomly structured individuals.

1. The population is evenly divided into parts: a total of *md*-1.
2. Half of each part of the population is produced by the grow method. The other half is produced using the full method. For the first part, the argument for the grow method, *m*, and the argument for the full method, *d*, is 2. For the second part 3 is used. This continues to part *md*-1, where the number *md* is used. Thus a population is created with good variation, utilizing both grow and full methods.

4.4.3 Fitness Test

Once the initial random population has been created, the individuals need to be assessed for their fitness. This is a problem specific issue that has to answer the question "how good (or bad) is this individual?" It is important that the fitness test gives as detailed an answer a possible, thus a test that answered "yes" or "no" would not be appropriate. This problem specific fitness value is given the term raw fitness—in this case, the greater the value the better the individual. For a ball-follower a good fitness test could be measured by the sum of the distances between the robot and the ball for a number of discrete times. In this case a smaller value indicates a better individual.

4.4.4 Functions and Terminals

The terminal and function sets are also important components of genetic programming. The terminal and function sets are the alphabet of the programs to be made. The terminal set consists of the variables and constants of the programs. The function set consists of the functions of the program. The functions are several mathematical functions, such as addition, subtraction, division, multiplication and other more complex functions.

4.4.5 The Genetic Operations

Having applied the fitness test to all the individuals in the initial random population, the evolutionary process starts. Individuals in the new population are formed by two main methods: *reproduction* and *crossover*. Once the new population is complete (i.e. the same size as the old) the old population is destroyed.

Reproduction

An asexual method, reproduction is where a selected individual copies itself into the new population. It is effectively the same as one individual surviving into the next generation. Generally 10% of the population is allowed to reproduce. If the fitness test does not change, reproduction can have a significant effect on the total time required for GP because a reproduced individual will have an identical fitness score to that of its parent. Thus a reproduced individual does not need to be tested, as the result is already known. This represented a 10% reduction in the required time to fitness test a population. However, a fitness test that has a random component, which is effectively a test that does not initialize to exactly the same starting scenario, would not apply for this increase in efficiency. The selection of an individual to undergo reproduction is the responsibility of the *selection function*.

4.4.6 Selection Functions

These methods would result in only a random search if it were not for the selection function. A number of selection functions can be used among which fitness proportionate selection, greedy over-selection, and tournament selection being the most common.

Fitness-proportionate Selection

With fitness-proportionate selection individuals are selected depending on their ability compared to the entire population, thus the best individual of a population is likely to be selected more frequently than the worst. The probability of selection is calculated with the following algorithm:

1. The raw fitness is restated in terms of *standardized fitness*. A lower standardized fitness value implies a better individual. If the raw fitness increases as an individual improves then an individual's standardized fitness is the maximum raw fitness (i.e. the fitness of the best individual in the population) minus the individual's raw fitness. If the raw fitness decreases as an individual improves, standardized fitness for an individual is equal to the individual's raw fitness.
2. Standardized fitness is then restated as *adjusted fitness*, where a higher value implies better fitness. The formula used for this is:

$$adj(i) = \frac{1}{1 + std(i)}$$

where $adj(i)$ is the adjusted fitness and $std(i)$ is the standardized fitness for individual i. The use of this adjustment is beneficial for separation of individuals with standardized fitness values that approach zero.

3. *Normalized fitness* is the form used by both selection methods. It is calculated from adjusted fitness in the following manner:

$$norm(i) = \frac{adj(i)}{\sum_{k=1}^{M} adj(k)}$$

where $norm(i)$ is the normalized fitness for individual i, and M is the number of individuals in the population.

4. The probability of selection (sp) is:

$$sp(i) = \frac{norm(i)}{\sum_{k=1}^{M} norm(k)}$$

This can be implemented by:

(a) Order the individuals in a population by their normalized fitness (best at the top of the list)
(b) Chose a random number, r, from zero to one.
(c) From the top of the list, loop through every individual keeping a total of their normalized fitness values. As soon as this total exceeds r stop the loop and select the current individual.

Greedy Over-selection

To reduce the number of generations required for a GP run, greedy over-selection is used. Again, individuals are selected based on their performance but this method biases selection towards the highest performers. Every individual is assessed, and their normalised fitness value calculated.

1. Using the normalised fitness values, the population is divided into two groups. Group I includes the top 20% of individuals while Group II contains the remaining 80%.
2. Individuals are selected from Group I 50% of the time. The selection method inside a group is fitness-proportionate.

Tournament Selection

In tournament selection pairs of individuals are chosen at random and the most fit one of the two is chosen for reproduction. This is meant to simulate the kind of competition that occurs in reproduction, and means that if two fairly unfit individuals are paired against each other, one of them is guaranteed to reproduce. A third way of choosing individuals is to rank them using the evaluation function and choose the ones at the top of the ranking, which means that only the best will be chosen for reproduction. As with most AI applications, it's a question of trying out different approaches to see which works for a particular problem.

4.4.7 Crossover Operation

Two primary operations exist for modifying structures in genetic programming. The most important one is the crossover operation. In the crossover operation, two solutions are sexually combined to form two new solutions or offspring. The parents are chosen from the population by a function of the fitness of the solutions. Three methods exist for selecting the solutions for the crossover operation. The first method uses probability based on the fitness of the solution. If $f(s_i(t))$ is the fitness of the solution s_i and $\sum_{j=1}^{M} f(s_j(t))$ is the total sum of all the members of the population, then the probability that the solution s_i will be copied to the next generation is:

$$\frac{f(s_i(t))}{\sum_{j=1}^{M} f(s_j(t))}$$

Another method for selecting the solution to be copied is tournament selection. Typically the genetic program chooses two solutions random. The solution with the higher fitness will win. This method simulates biological mating patterns in which, two members of the same sex compete to mate with a third one of a different sex. Finally, the third method is done by rank. In rank selection, selection is based on the rank, (not the numerical value) of the fitness values of the solutions of the population.

The creation of the offsprings from the crossover operation is accomplish by deleting the crossover fragment of the first parent and then inserting the crossover fragment of the second parent. The second offspring is produced in a symmetric manner. For example consider the two S-expressions in Figure 4.5, written in a modified scheme programming language and represented in a tree.

Given a tree based GP the crossover operator works in the following manner:

1. Select two individuals, based on the selection mechanism.
2. Select a random subtree in each parent.
3. Swap the two subtrees or sequences of instructions.

The crossover operator closely mimics the process of biological sexual reproduction. Based on this notion, the crossover operator has been used to claim that GP search is more efficient than methods based solely on mutation. argued that a population in a GP system contains *building blocks*; a building block can be any tree or subtree which is present within a fraction of the population. This hypothesis follows the same line of argument as the *Building block hypothesis* from genetic algorithms.

Given the building blocks and crossover the systems should be able to combine "good" building blocks to more fit individuals, and small building block should be able to combine into larger building blocks, hence, making the GP search more efficient than other methods.

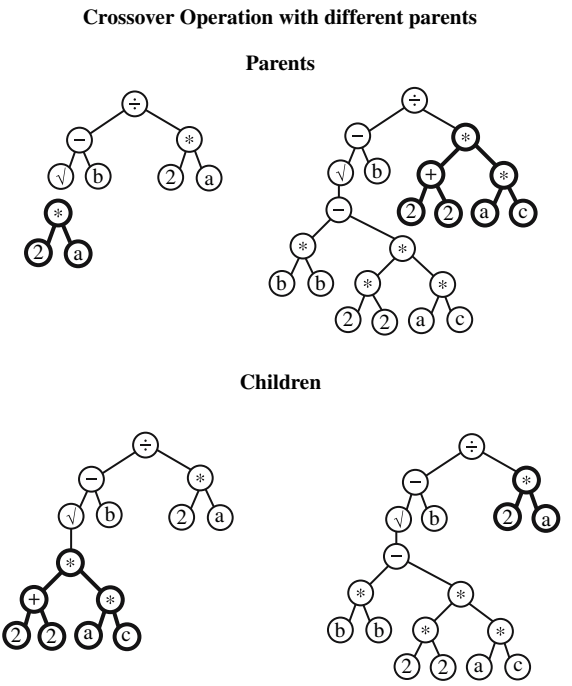


Fig. 4.5 Crossover operation for genetic programming. The bold selections on both parents are swapped to create the offspring or children

An important improvement that genetic programming displays over genetic algorithms is its ability to create two new solutions from the same solution. In the Figure 4.6 the same parent is used twice to create two new children.

4.4.8 Mutation

Mutation is another important feature of genetic programming. Two types of mutations are possible. In the first kind a function can only replace a function or a terminal can only replace a terminal. In the second kind an entire subtree can replace another subtree. Figure 4.7 explains the concept of mutation:

4.4.9 User Decisions

The user must make a number of decisions before the GP system may begin. Firstly, the available genes need selecting and creating. Secondly, the user must specify a number of control parameters. The decisions are critically important as they have a limiting effect on the search space of possible programs. Too great a limit may remove all chance of evolving an acceptable individual.

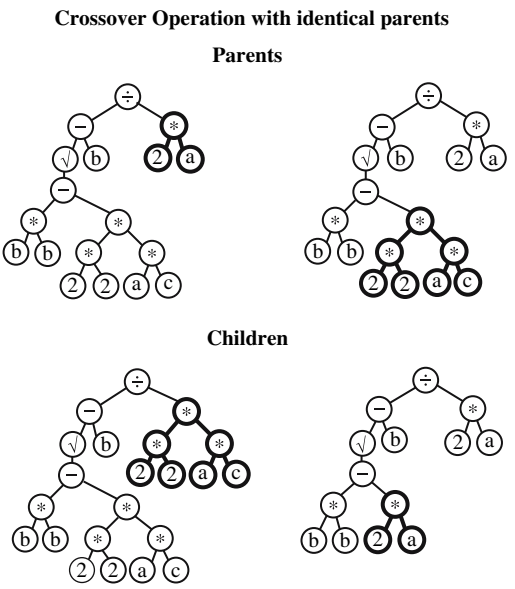


Fig. 4.6 This figure illustrates one of the main advantages of genetic programming over genetic algorithms. In genetic programming identical parents can yield different offspring, while in genetic algorithms identical parents would yield identical offspring. The bold selections indicate the subtrees to be swapped

The selection of genes to perform a task is important. It is important to include every necessary function and terminal. However, selecting more than the minimum required set of genes poses little problem as the genes that are not useful will be bred out of the population. The control parameters that need to be set are:

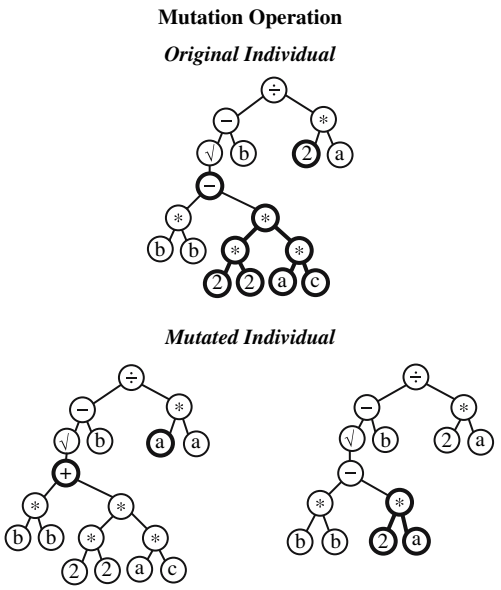


Fig. 4.7 Two different types of mutations. The top parse tree is the original agent. The bottom left parse tree illustrates a mutation of a single terminal (2) for another single terminal (a). It also illustrates a mutation of a single function $(-)$ for another single function $(+)$. The parse tree on the bottom right illustrates the replacement of a subtree by another subtree

1. Population size: A larger population allows for a greater exploration of the problem space at each generation and increases the chance of evolving a solution. In general, the more complex a problem the greater the population size needed.
2. Maximum number of generations: The evolutionary process needs to be given time; the greater the maximum number of generations the greater the chance of evolving a solution. However, further evolution of a population does not guarantee a solution will be found—it may be better to start again with a different initial population. So if, after a user-defined number of generations, a sufficiently successful individual has not evolved then the process should halt.
3. Probability of crossover: The proportion of the population that will undergo crossover before entering the new population has to be determined. In general, this value varies from 0.90–90% of the population that undergoes crossover.
4. Probability of reproduction: The proportion of individuals in a population that will undergo reproduction is generally 0.10–10%.

4.5 An Illustration

The Boolean 11-multiplexer problem is described here. The problem is to design a Boolean function (or circuit) which implements a Boolean 11-multiplexer. The multiplexer has 11 inputs and one output. The first three inputs, `a0, a1`, and `a2`, can be considered as address lines. They describe the binary representation of an integer between 0 and 7. This integer chooses one of the 7 remaining inputs, which are labeled `d0, d1, d2, d3, d4, d5, d6, d7`. The correct output for the multiplexer is the input on the line specified by the address lines. In Figure 4.8, the

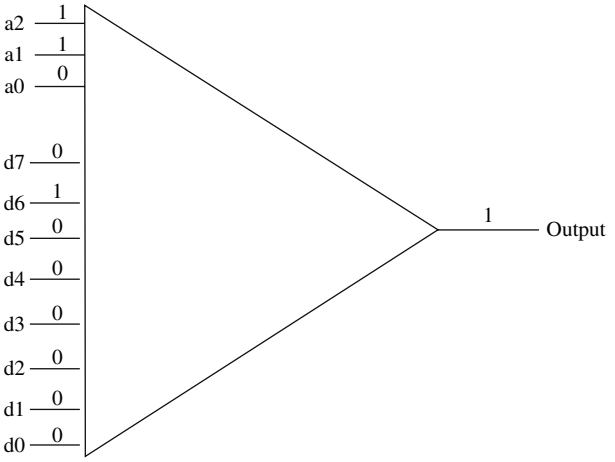


Fig. 4.8 Boolean 11-multiplexer with input of 11001000000 and output of 1

address lines specify the integer 6, so the output should be the input on line d6, which is 1.

There are $2^{11} = 2048$ possible inputs to the function, so all can be tested. The raw fitness is the number of cases for which the output is correct. The standardized fitness is 2048 minus the raw fitness. The terminals are the inputs to the function. The non-terminals are AND, OR, NOT, IF.

A population size of 4000 was used with over selection in this problem. In the run, the solution was discovered in generation 9. This solution in Lisp notation is:

```
(IF A0 (IF A2 (IF A1 D7 (IF A0 D5 D0))
         (IF A0 (IF A1 (IF A2 D7 D3) D1) D0))
       (IF A2 (IF A1 D6 D4)
              (IF A2 D4 (IF A1 D2 (IF A2 D7 D0)))))
```

4.6 The GP Paradigm in Machine Learning

In areas such as artificial intelligence, machine learning, or symbolic processing, many problems arise whose resolution can be considered as the search of a computer program, inside a space of possible programs that produce some desired outputs from the inputs. This search should be carried out in such a way that the searched program be the more adequate to the problem that is considered. The genetic programming (GP) paradigm provides the appropriate framework to develop this type of search in an efficient and versatile way, since it adapts to any problem type, i.e., it is very robust.

The paradigm of genetic programming is based on the principle of survival of fittest (C. Darwin). Starting from a randomly-generated initial population, it evolves populations following this principle. The new individuals are a product of genetic operations on the current population's better individuals. In connection with the genetic algorithms (GAs), GP shares with these the philosophy and the characteristics of being heuristic and stochastic.

Inside the area of the machine learning, several paradigms are focused toward the resolution of problems. In each paradigm the used structures are different:

Connectionist Model

The solution to the problem is given as a group of real-valued weights that indicate if the value of the signal that goes by a certain connection of a neural network is amplified or diminished.
Application field: NEURAL NETWORKS

Evolutionary Model

The solutions are fixed-length strings (in the classic model). Each chromosome represents a possible solution to the problem. A conventional genetic algorithm is applied to obtain the best solution (or a good enough solution) among all possible solutions.
Application field: GENETIC ALGORITHMS

Inductive Model

According to this paradigm the solutions to a certain problem are given by decision trees. Each one of these trees classifies each instance of the problem in classes, for which a possible solution exists.
Application field: CLASSIFIER SYSTEMS

Each one of the mentioned models can be more or less effective solving a certain problem type. The approach that is used to determine the efficiency of a method is, in the first place, the flexibility to adapt to several types of problems, and in second its easiness to represent the solutions to this problem in a natural and comprehensible way for the user. Computer programs offer flexibility:

- to perform operations with variables of different types.
- to carry out operations conditioned to the results obtained in intermediate points.
- to carry out iterations and recursions.
- to define routines that can be used later on.

This idea of flexibility includes the concepts of flexibility in the size, the form and the structural complexity of the solution. That is to say, the user should avoid stating any type of explanation or previous imposition on the size or form of the solution. The true power of GP resides in its capacity of adaptation to the problem, for what the considerations on the size, the complexity or the form of the solution should emerge during the own resolution process. The importance of the representation of the solutions resides in that the genetic algorithms manipulate the structure of this representation directly and not the own solution.

String-representations do not directly provide the hierarchical structure of programs. Also, the adaptation of the form, size and complexity of individuals becomes very difficult. Therefore, GP has spread toward more complex representations that contribute the needy flexibility. Smith introduces the strings of variable length and IF-THEN-ELSE components.

Genetic algorithms are used in a wide range of applications thanks to the adaptation of their evolutionary structures. For example as classification systems, as the system designed by Wilson for the classification of boolean functions. Goldberg introduced the Messy genetic algorithms that handle populations of individuals represented by strings of characters of variable length.

GP uses programs like individuals, but the division of opinions appears with its implementation. Cramer uses a parse-tree-like representation of the code, and defines suitable operations (for example, exchange of subtrees for recombination).

Fujiki and Dickinson have developed a system that is based on the generation of programs that use simple conditional sentences of LISP (COND).

The resolution of many problems can be modelled as an evolutionary process in which the fittest individual survives. The simulation of this evolutionary process begins with the generation of an initial population, composed by computer programs that represent the individuals. These programs are generated starting from the group of functions and terminal elements that adapt better to the problem to solve. In most cases, the election of the group of terminal and non terminals (functions) is critical to make the algorithm works properly.

For example, to generate complex mathematical functions it is interesting to introduce in the group of non terminals such functions as sines, cosines, and so on. For graphic applications, it is usually quite normal to introduce primitive of the type Line, Circle, Torus that graphically represent figures in diverse ways.

The appropriateness of each program is measured in terms of how of well it performs in the environment of the particular problem. This measure is denominated fitness. A program is generally evaluated in different representative cases, and the final measure of its fitness will be an average of all the measures (one for each case). Usually, the result of the genetic algorithm is the best individual generated in generation n, being n the maximum number of generations.

4.7 Preparatory Steps of Genetic Programming

The human user communicates the high-level statement of the problem to the genetic programming system by performing certain well-defined preparatory steps. The five major preparatory steps for the basic version of genetic programming require the human user to specify

(1) the set of terminals (e.g., the independent variables of the problem, zero-argument functions, and random constants) for each branch of the to-be-evolved program,
(2) the set of primitive functions for each branch of the to-be-evolved program,
(3) the fitness measure (for explicitly or implicitly measuring the fitness of individuals in the population),
(4) certain parameters for controlling the run, and
(5) the termination criterion and method for designating the result of the run.

4.7.1 The Terminal Set

The terminal set contains all the variables and constants which will appear in the evolved programs. This will typically include some numbers which may be randomly generated, and, as with the function set, it will include problem specific details. In a robot controlling scenario, it may be that movement functions are parameterized by directions such as left, right, forward, backwards, which would form

part of the terminal set for that GP application. Similarly, in a graphics application, constants such as *pi* might be put into the terminal set. The terminal set is so called, because in the tree representations, the constants and variables are found at the end of the branches.

4.7.2 The Function Set

A set of domain specific functions used in conjunction with the terminal set to construct potential solutions to a given problem. For symbolic regression this could consist of a set of basic mathematical functions, while Boolean and conditional operators could be included for classification problems. As with the evaluation function, the set of functions will be hand-carved for the particular task. For instance, while evolving a program to control a robot as it tries to find its way out of a maze, the functions will include things like turning left, turning right, going forward, etc. If functions are evolved to manipulate images, the functions will involve mathematical functions such as sine and cosine, and pixel functions, such as finding pixel colours, hues and intensities, and setting pixel colours, hues and intensities. The function set also includes the set of programmatic functions such as if-then-else and for-loops. It is often instructive to see if good programs can be evolved with, for example, while-loops but no for-loops.

4.7.3 The Fitness Function

Fitness is a numeric value assigned to each member of a population to provide a measure of the appropriateness of a solution to the problem in question.

4.7.4 The Algorithm Control Parameters

This includes the population size and the crossover and mutation probabilities. There are many possibilities for how the search will proceed, and the user should tweak various parameters to optimise the performance of the GP engine. The main consideration will be the size of the population, as this will effect the GP the most: larger populations will mean fewer generations in the time available, but will mean larger diversity within the population of programs. Given the programs will grow as they evolve, another important parameter will be a cap on the length of the programs that can be produced. One criticism of GP approaches is that the programs produced are too large and complicated to be understood, so, if being able to understand the resulting programs is a consideration, the length of the programs should be kept relatively small. Other parameters will control various probabilities, including the probability that each genetic operator (see later) will be employed.

4.7.5 The Termination Criterion

The ways to specify when the GP engine should stop are very similar to those for Genetic Algorithms. One possibility is to let the process run for a certain amount of time, or until it has produced a certain number of generations, then take the best individual produced in any generation. Many GP implementations enable the user to monitor the process and click on the stop button when it appears that the fitness of the individuals has reached a plateau.

This is generally a predefined number of generations or an error tolerance on the fitness. It should be noted that the first 3 components determine the algorithm search space, while the final 2 components affect the quality and speed of search. In order to further illustrate the coding procedure and the genetic operators used for GP, a symbolic regression example will be used. Consider the problem of predicting the numeric value of an output variable, y, from two input variables a and b. One possible symbolic representation for y in terms of a and b would be,

$$y = (a - b)/3 \tag{4.1}$$

Figure 4.9 demonstrates how this expression may be represented as a tree structure

With this tree representation, the genetic operators of crossover and mutation must be posed in a fashion that allows the syntax of resulting expressions to be preserved. A valid crossover operation is shown where the two parent expressions are given by:

Parent 1: $y = (a - b)/3$

Parent 2: $y = (c - b)^*(a + c)$

Parent 1 has input variables '*a*' and '*b*' and a constant '*3*' while parent 2 has three input variables '*a*', '*b*' and '*c*'.

Both expressions attempt to predict the process output, '*y*'.

If the '/' from parent 1 and the '*' from parent 2 are chosen as the crossover points, then the two offspring are given by:

Offspring 1: $y = (a - b)/(a + c)$

Offspring 2: $y = (c - b)^*3$

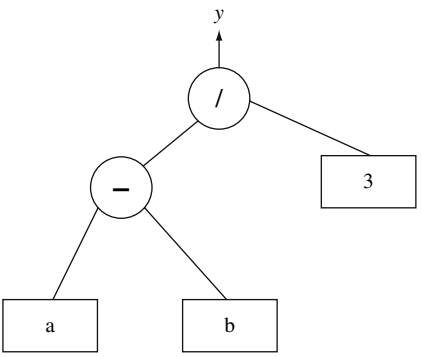


Fig. 4.9 Representation of a numeric expression using a tree structure

4.8 Flow – Chart of Genetic Programming

The main difference between genetic programming and genetic algorithms is the representation of the solution. Genetic programming creates computer programs in the lisp or scheme computer languages as the solution. Genetic algorithms create a string of numbers that represent the solution. Genetic programming uses four executional steps to solve problems:

1) Generate an initial population of random compositions of the functions and terminals of the problem (computer programs).
2) Execute each program in the population and assign it a fitness value according to how well it solves the problem.
3) Create a new population of computer programs.

   (i) Copy the best existing programs.
   (ii) Create new computer programs by mutation.
   (iii) Create new computer programs by crossover (sexual reproduction).

4) The best computer program that appeared in any generation, the best-so-far solution, is designated as the result of genetic programming.

Genetic programming is problem-independent in the sense that the flowchart specifying the basic sequence of executional steps is not modified for each new run or each new problem. There is usually no discretionary human intervention or interaction during a run of genetic programming (although a human user may exercise judgment as to whether to terminate a run).

Genetic programming starts with an initial population of computer programs composed of functions and terminals appropriate to the problem. The individual programs in the initial population are typically generated by recursively generating a rooted point-labeled program tree composed of random choices of the primitive functions and terminals. The initial individuals are usually generated subject to a pre-established maximum size. In general, the programs in the population are of different size (number of functions and terminals) and of different shape (the particular graphical arrangement of functions and terminals in the program tree).

Each individual program in the population is executed. Then, each individual program in the population is either measured or compared in terms of how well it performs the task at hand. For many problems, this measurement yields a single explicit numerical value, called *fitness*. The fitness of a program may be measured in many different ways, including, for example, in terms of the amount of error between its output and the desired output, the amount of time (fuel, money, etc.) required to bring a system to a desired target state, the accuracy of the program in recognizing patterns or classifying objects into classes, the payoff that a game-playing program produces, or the compliance of a complex structure (such as an antenna, circuit, or controller) with user-specified design criteria. The execution of the program sometimes returns one or more explicit values. Alternatively, the execution of a program may consist only of side effects on the state of a world (e.g., a

robot's actions). Alternatively, the execution of a program may produce both return values and side effects.

The fitness measure is, for many practical problems, multiobjective in the sense that it combines two or more different elements. The different elements of the fitness measure are often in competition with one another to some degree. For many problems, each program in the population is executed over a representative sample of different *fitness cases*. These fitness cases may represent different values of the program's input(s), different initial conditions of a system, or different environments. Sometimes the fitness cases are constructed probabilistically. The creation of the initial random population is, in effect, a blind random search of the search space of the problem. It provides a baseline for judging future search efforts. Typically, the individual programs in generation 0 all have exceedingly poor fitness. Nonetheless, some individuals in the population are (usually) more fit than others. The differences in fitness are then exploited by genetic programming. Genetic programming applies Darwinian selection and the genetic operations to create a new population of offspring programs from the current population.

The genetic operations include crossover (sexual recombination), mutation, reproduction, and the architecture-altering operations. These genetic operations are applied to individual(s) that are probabilistically selected from the population based on fitness. In this probabilistic selection process, better individuals are favored over inferior individuals. However, the best individual in the population is not necessarily selected and the worst individual in the population is not necessarily passed over.

After the genetic operations are performed on the current population, the population of offspring (i.e., the new generation) replaces the current population (i.e., the now-old generation). This iterative process of measuring fitness and performing the genetic operations is repeated over many generations. The run of genetic programming terminates when the termination criterion (as provided by the fifth preparatory step) is satisfied. The outcome of the run is specified by the method of result designation. The best individual ever encountered during the run (i.e., the best-so-far individual) is typically designated as the result of the run.

All programs in the initial random population (generation 0) of a run of genetic programming are syntactically valid, executable programs. The genetic operations that are performed during the run (i.e., crossover, mutation, reproduction, and the architecture-altering operations) are designed to produce offspring that are syntactically valid, executable programs. Thus, every individual created during a run of genetic programming (including, in particular, the best-of-run individual) is a syntactically valid, executable program.

There are numerous alternative implementations of genetic programming that vary from the foregoing brief description.

Figure 4.10 below is a flowchart showing the executional steps of a run of genetic programming. The flowchart shows the genetic operations of crossover, reproduction, and mutation as well as the architecture-altering operations.

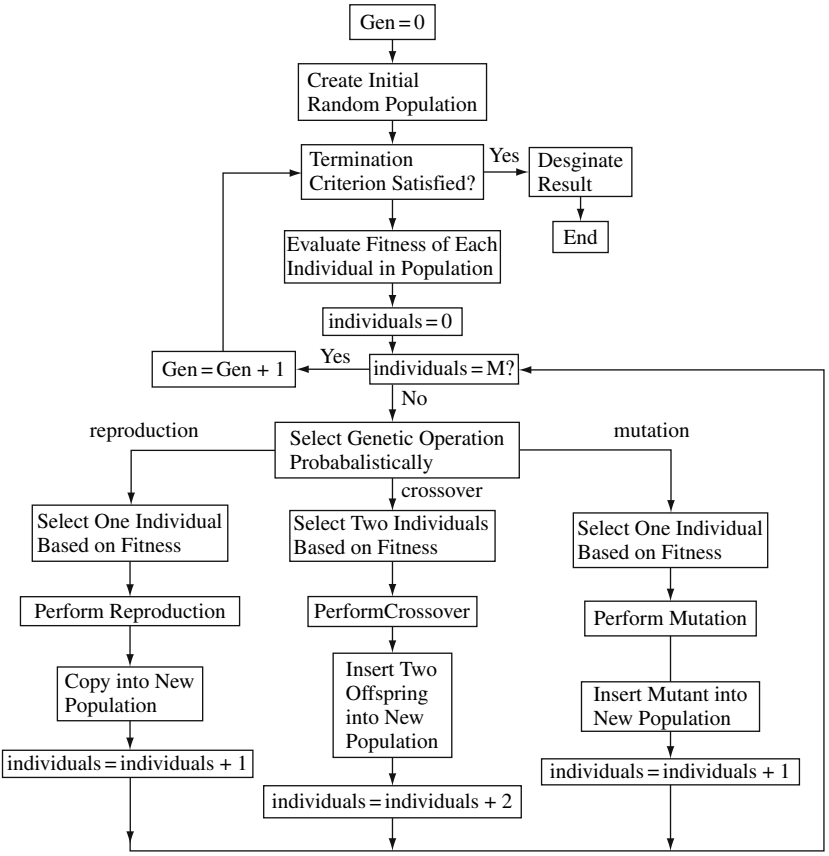


Fig. 4.10 Genetic programming flowchart

4.9 Type Constraints in Genetic Programming

Data structures are a key concept in computer programming. They provide a mechanism to group together data which logically belong together and to manipulate this data as a logical unit. Data typing (which is implemented in many computer languages including C++, Ada, Pascal and LISP) is a way to associate a type (or class) with each data structure instance to allow the code to handle each data structure differently based on its type. Even data structures with the same underlying physical structure can have different logical types; for example, a 2×3 matrix, a 6-vector, and an array of 3 complex number will likely have the same physical representation but will have different data types. Different programming languages use data typing differently. Strongly typed languages, such as Ada and Pascal, use data types at program generation time to ensure that functions only receive as arguments the particular data types they are expecting. Dynamically typed languages, such as

LISP, use data types at program execution time to allow programs to handle data differently based on their types.

Standard genetic programming is not designed to handle a mixture of data types. In fact, one of its assumptions is "closure", which states that any non-terminal should be able to handle as an argument any data type and value returned from a terminal or non-terminal. While closure does not prohibit the use of multiple data types, forcing a problem which uses multiple data types to fit the closure constraint can severely and unnecessarily hurt the performance of genetic programming on that problem. One way to get around the closure constraint is to carefully define the terminals and non-terminals so as to not introduce multiple data types. For example, a Boolean data type (which can assume only two values, true and false) is distinct from a real-valued data type. Koza avoids introducing Boolean data types using a number of tricks in his definitions of functions. First, he avoids using predicates such as FOODHERE and LESS-THAN, which return Boolean values, and instead uses branching constructs such as IF-FOOD-HERE and IF-LESS-THAN, which return real values (or, more precisely, the same types as the arguments which get evaluated). Second, like the C and C++ programming languages, Koza uses functions which treat real-valued data as Boolean, considering a subset of the reals to be "true" and its complement to be "false". For example, IFLTZ (If Less Than Zero) is an IF-THEN-ELSE construct which evaluates and returns its second argument if the first argument is less than zero and evaluates and returns its third argument otherwise. Note that Koza's partitioning of the reals is more likely to yield true branching (as opposed to consistent execution of a single subtree) than the C language practice of considering 0 to be false and everything else true.

However, this approach is limited in its applicability. For example, in the matrix/vector manipulation problems and list manipulation problems, there is no way to avoid introducing multiple data types. (For the former, we need both vectors and matrices, while for the latter we need both lists and list elements.) A way to simultaneously enforce closure and allow multiple data types is through the use of dynamic typing. In this approach, each non-terminal must handle as any of its arguments any data type that can be returned from any terminal or non-terminal. There are two (not necessarily exclusive) ways to do this. The first is to have the functions actually perform different actions with different argument types. The second is to have the functions signal an error when the arguments are of inconsistent type and assign an infinitely bad evaluation to this parse tree. The first approach, that of handling all data types, works reasonably well when there are natural ways to cast any of the produced data types to any other. For example, consider using two data types, REAL and COMPLEX. When arithmetic functions, such as $+$, $-$, and *, have one real argument and one complex argument, they can cast the real number to a complex number whose real part is the original number and whose complex part is zero. Comparison operators, such as IFLTZ, can cast complex numbers to reals before performing the comparison either by using the real portion or by using the magnitude. However, often there are not natural ways to cast from one data type to another. For example, consider trying to add a 3-vector and a 4×2 matrix. Consider the matrix to be an 8-vector, throw away its last five entries, and then

add. The problem with such unnatural operations is that, while they may succeed in finding a solution for a particular set of data, they are unlikely to be part of a symbolic expression that can generalize to new data (a problem which is demonstrated in our experiment). Therefore, it is usually best to avoid such "unnatural" operations and restrict the operations to ones which make sense with respect to data types. This bias against unnatural operations is an example of what in machine learning is called "inductive bias". Inductive bias is the propensity to select one solution over another based on criteria which reflect experience with similar problems. For example, in standard genetic programming, the human's choice of terminal and non-terminal sets and maximum tree size, provides a definite inductive bias for a particular problem. Ensuring consistency of data types, mechanisms for which is discussed below, it is an inductive bias which can be enforced without human intervention. There is a way to enforce data type constraints while using dynamic data typing, which is to return an infinitely bad evaluation for any tree in which data type constraints are violated. The problem with this approach is that it can be terribly inefficient, spending most of its time evaluating trees which turn out to be illegal.

A better way to enforce data type constraints is to use strong typing and hence to only generate parse trees which satisfy these constraints. This is essentially what Koza does with "constrained syntactic structures". For problems requiring data typing, he defines a set of syntactic rules which state, for each non-terminal, which terminals and non-terminals can be its children nodes in a parse tree. He then enforces these syntactic constraints by applying them while generating new trees and while performing genetic operations.

4.10 Enhanced Versions of Genetic Programming

Genetic programming is a powerful method for automatically generating computer programs via the process of natural selection. However, in its standard form, there is no way to restrict the programs it generates to those where the functions operate on appropriate data types. In the case when the programs manipulate multiple data types and contain functions designed to operate on particular data types, this can lead to unnecessarily large search times and/or unnecessarily poor generalization performance. These problems were overcome using enhanced versions of GP such as

a. Meta Genetic programming
b. Cartesian Genetic Programming
c. Strongly Typed Genetic Programming

4.10.1 Meta-genetic Programming

In genetic programming the genetic operators are usually fixed by the programmer. Typically these are a variety of crossover and propagation, but can also include others, for example mutation. A variety of alternative operators has been invented and investigated. It is usually found that, at least for some problems these perform better than the 'classic' mix of mostly tree-crossover and some propagation.

Meta-Genetic Programming (MGP) encodes these operators as trees. These "act" on other tree structures to produce the next generation. This representation allows the simultaneous evolution of the operators along with the population of solutions. In this technique what is co-evolved along with the base population is not a fixed set of parameters or extra genetic material associated with each gene, by operators who are themselves encoded as variable length representations – so their form as well as their preponderance can be evolved. This technique introduces extra computational cost, which must be weighed against any advantage gained. Also the technique turns out to be very sensitive to biases in the syntax from which the operators are generated, it is thus much less robust.

Iterating this technique can involve populations of operators acting on populations of operators acting on populations etc., and even populations acting on themselves. This allows a range of techniques to be considered within a common framework - even allowing the introduction of deductive operators such as Modus Ponens to be included.

The Basic Technique

Instead of being hard-coded, the operators are represented as tree structures. For simplicity this is an untyped tree structure. Two randomly chosen trees along with randomly chosen nodes from those trees are passed to be operated on by the operator tree. The terminals refer

1. For a survey, which covers such approaches to one or other of these trees at the respective node. The structure of the operator tree represents transformations of this (tree, node) pair until the root node produces the resulting gene for the next operator.

Thus there could be the following terminals:

rand1 - return the first, randomly chosen gene plus random node;
rand2 - return the second, randomly chosen gene plus random node;
bott1 - return the first, randomly chosen gene plus the root node;
bott2 - return the second, randomly chosen gene plus the root node.

and the following branching nodes:

top - cut the passed tree at the passed node and return it with the root node;
up1 - pass the tree as it is but choose the node immediately up the first branch
up2 - pass the tree as it is but choose the node immediately up the second branch
down - pass the tree as it is but choose the node immediately down
down1 - identical to **down** (so that the operators will not have an inherent bias up or down);
subs - pass down a new tree which is made by substituting the subtree at the pass down a new tree which is made by substituting the subtree at the indicated node

For example if the operator, **[up2 [top [rand1]]]** were passed the following input pair of (tree, node) pairs to act upon, `([B [A [A [B [2]] [1]] [3]]], [1 1]), ([A [1] [2]], [2])` it would return the pair `([A [B [2]], [2])` – this process is illustrated below in Figure 4.11.

A second example is if the operator, **[subs [bott2] [rand2]]** were passed the same inputs it would return `([A [1] [A [1] [2]]], [2])` – as illustrated in Figure 4.12.

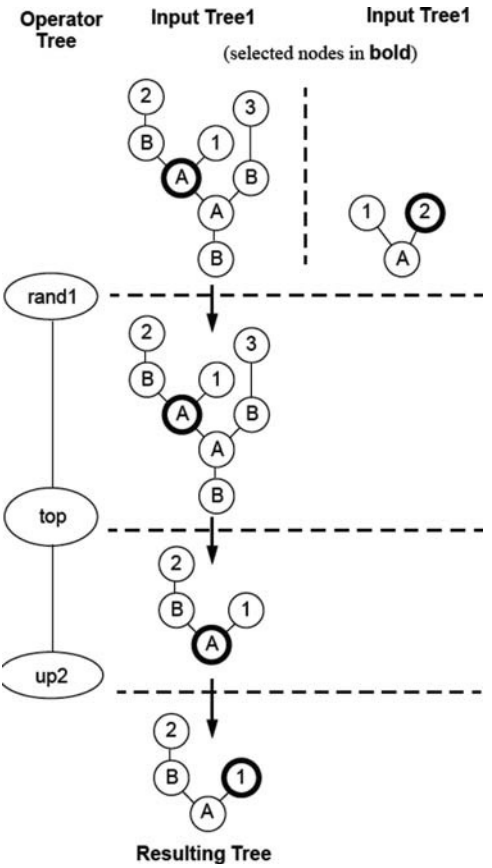


Fig. 4.11 An example action of an operator on a pair of trees with selected nodes

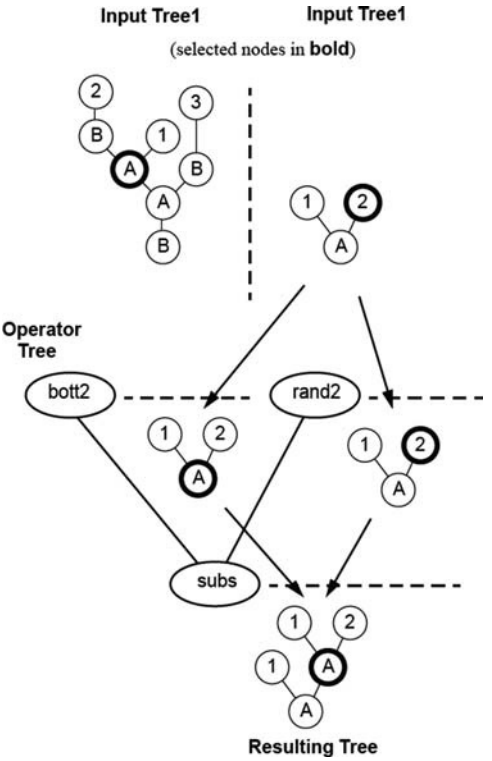


Fig. 4.12 An operator acting on a pair of trees with selected nodes

Thus a cut & graft operator (one half of a traditional GP crossover) could be represented as shown in Figure 4.13

This takes the first random gene, cuts off the top at the randomly chosen position and then substitutes this for the subtree at the randomly chosen position of the second chosen gene. The single leaf **rand1** would represent propagation. The population of operators is treated just like any other genetic population, being itself operated on by another population of operators. This sequence must eventually stop

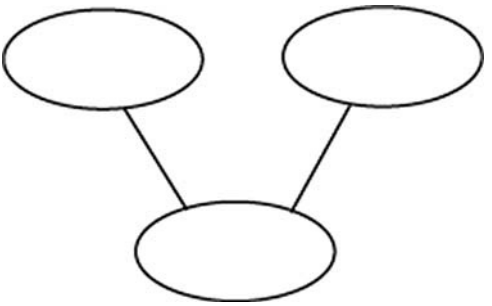
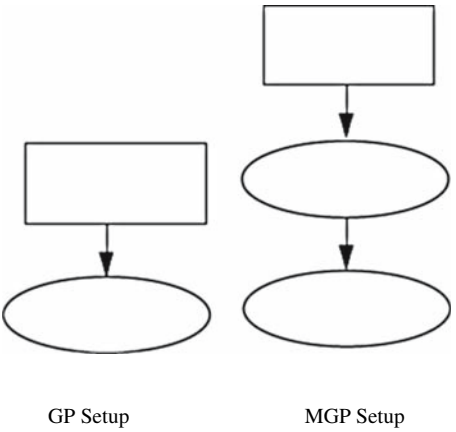


Fig. 4.13 A cut and graft operator encoded as a tree

Fig. 4.14 GP and MGP setups



with a fixed non-evolving population of operators or a population that acts upon itself.

The base population and operator population(s) will have different fitness functions - the basic one determined by the problem at hand and the operator's function by some measure of success in increasing the fitness of the population they operate on.

MGP as a Framework for a Set of Techniques

The relation of "Population A acting as operators on population B" can be represented diagrammatically by an arrow. In Figure 4.14, traditional GP and basic MGP setups are illustrated, with evolving population represented by ellipses and fixed populations by rectangles.

There is no a priori reason why many different setups should not be investigated. For example Figure 4.15 shows a simple enhancement of GP where just the proportions of the operators is changed, implementing a similar technique as invented by for GAs, and Figure 4.16 illustrates the possibility of combining a fixed and evolving population of operators.

Finally there is no reason why populations of operators should not operate recursively. One can even imagine situations in which the interpretation (i.e. fitness) of the base population and operator population was done consistently enough such that you only had *one* population acting on itself in a self-organising loop, as in Figure 4.17. In this case care would need to be taken in the allocation of fitness values, maybe by some mechanism as the bucket-brigade algorithm or similar. Such a structure may allow the implicit decision of the best structure and allow for previously unimaginable hybrid operator-base genes. It does seem unlikely, however, that the same population of operators would be optimal for evolving the operators as the base population, due to their different functions.

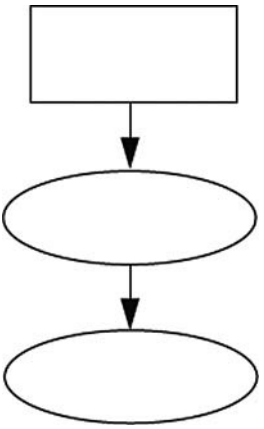


Fig. 4.15 GP with the proportion of operators evolving

MGP can be compared to approaches in GAs, such as in which augment a standard GA with a series of crossover masks each of which has a weight associated, which changes according to their success, and which af fects their later utilisation. In MGP, however, there are a potentially infinite number of such operators Much of the power of GP comes from its low computational cost compared to its effectiveness, allowing large populations to successfully evolve solutions where more carefully directed algorithms have failed. Thus the conditions of application of MGP are very important - when it is useful to use and when not.

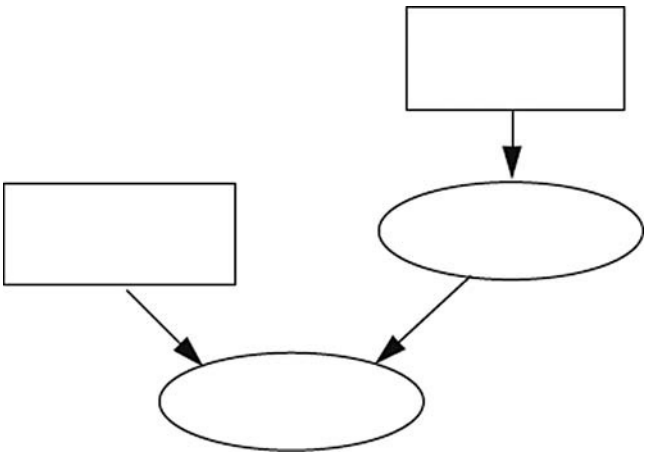


Fig. 4.16 A setup with a mixture of fixed and evolving operators

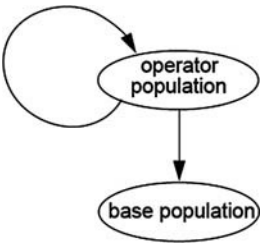


Fig. 4.17 A recursively evolving MGP setup

4.10.2 Cartesian Genetic Programming

In its original form Genetic Programming (GP) has evolved programs in the form of LISP parse trees. Usually large populations are used and crossover is used as the primary method of developing new candidate solutions from older programs. In contrast to this in Evolutionary Programming has tended to emphasize the importance of mutation operators. Although GP and EP place different emphasis on the evolutionary operators they both tend to share the representation of programs as parse trees in which there is no distinction between genotype and phenotype. Trees are a special form of graphs in which two nodes must have at most one path between them (a *path* is a sequence of connected nodes). One of the original motivations for a tree-based approach was to allow a solution to the problem of applying crossover to variable length genotypes. A new approach of GP - *Cartesian Genetic Programming (CGP)* where the genotype is represented as a list of integers that are mapped to directed graphs rather than trees. One motivation for this is that it uses graphs which are more general than trees. However the original motivation came from the effectiveness of the approach in learning Boolean functions where it proved to be considerably more efficient than standard GP methods. In CGP the genotypes are of fixed length but the phenotypes are of variable length according to the number of unexpressed genes. In Binary Genetic Programming (BGP) binary strings are translated and repaired to form valid programs.

In CGP no repair is necessary. Another potential advantage of employing a genotype-phenotype mapping is that it allows for the possibility of many genotypes mapping to the same phenotype and so explicitly allows *neutrality* to be present. Neutrality refers to the presence of genotypes with the same fitness. The importance of neutrality is widely recognized in modern theories of natural molecular evolution. It is important to note that neutrality arising from genotype redundancy can only be useful when the neutral changes are likely to alter the potential effects of future genotypic change, merely adding unexpressed genes will not facilitate useful neutral evolution. In CGP there are very large number of genotypes that map to identical genotypes due to the presence of a large amount of redundancy. Firstly there is *node redundancy* that is caused by genes associated with nodes that are not part of the connected graph representing the program. This redundancy is very large at the beginning of the evolutionary run as many nodes are not connected in the early populations. The node redundancy gradually reduces during the run to a level

that is determined by average number of nodes required to implement a satisfactory program and the maximum allowed number of nodes. Another form of redundancy in CGP, also present in all other forms of GP is, *functional redundancy*. In this case, a number of nodes implement a sub-function that actually may be implemented with fewer nodes. This growth in the number of redundant nodes constitutes *bloat*. The third form of redundancy, called *input redundancy*, occurs when some node functions are not connected to some of the input nodes. For example, it is seen that all the nodes used to evolve programs to solve a problem having three inputs and one output, despite the fact that only one function uses the three inputs. Node and input redundancy both have the potential of adding useful neutrality. An unconnected node may undergo neutral change and later become connected – this might be necessary to achieve a higher fitness. A node with a redundant input might, after a mutation that altered the arity of the node function, suddenly become useful. Functional redundancy probably positively contributes to the evolvability of the target function or program as it increases the number of ways that it might be built. The possibility of disconnecting nodes that CGP allows might have a useful role in keeping the attendant bloat in check. Here CGP is introduced as an alternative methodology to standard GP, and secondly, to extend CGP to non-Boolean problems and show that it is a useful method for evolving programs with other data types. It is important that researchers in Genetic Programming explore the advantages and disadvantages of many representations rather than confining themselves to one dominant form. Diversity is important in research just as it is in evolving populations.

Methodology of CGP

CGP is Cartesian in the sense that the method considers a grid of nodes that are addressed in a Cartesian coordinate system. CGP has some of points of similarity with Parallel Distributed Genetic Programming (PDGP) and the graph-based GP system PADO (Parallel Algorithm Discovery and Orchestration). In the former graphs were evolved *without* the use of a genotype-phenotype mapping and various sophisticated crossover operators were defined. In the latter, each program is represented as an arbitrary directed graph of N nodes, where each node may have up to N outputs. Moreover, in PADO each node possesses its own private stack based memory, and also access to a globally defined indexed memory. A Cartesian program (CP) denoted P is defined as a set $\{G, n_i, n_o, n_n, F, n_f, n_r, n_c, l\}$ where G represents the genotype and is itself a set of integers representing the indexed n_i program inputs, the n_n node input connections and functions, and the n_o program output connections. The set F represents the n_f functions of the nodes. The number of nodes in a row and column are given by n_r, n_c respectively. Finally the program interconnectivity is defined by the levels back parameter l, which determines how many previous columns of cells may have their outputs connected to a node in the current column (the primary inputs are treated in the same way as node outputs). Basically only feed-forward connectivity is considered. However a Cartesian program can be

0 1 3 *0* 1 0 4 *1* 3 5 2 *2* 8 5 0 *1* 6 8 3 *2* 4 2 6 *0*
6 10 7 *2* 9 11 10 *1* 8 8 6 *2* 9 10 13 *15* 13 14 11 *16* 10 10 14 *2* 13 16 17

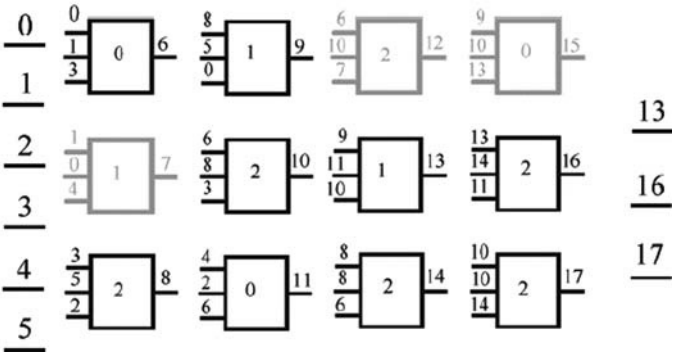


Fig. 4.18 Genotype-phenotype mapping. Phenotype (above). Genotype (below). For a program with six inputs and 3 outputs, and three functions (0, 1, 2 inside square nodes, in italics in genotype). The grey squares indicate unconnected nodes

readily extended to include sequential processes (e.g. loops), by allowing the inputs to nodes to be connected to the program outputs through a clock. Note that the program inputs are allowed to connect to any node input. An example of the genotype and genotype-phenotype mapping is depicted in Figure 4.18 for a program with six inputs and three outputs. The example shows a feed-forward program of three rows by four columns, with $l = 2$ and $n_n = 3$.

Nodes in the same column are not allowed to be connected to each other, and any node may be either connected or disconnected. The CP with the most freely connected network is obtained when the $n_r = 1$ *and* $l = n_c$, i.e. any node can be connected to any node on the right. The length of the genotype G that maps to a particular CP is fixed and is equal to $n_r\ n_c\ (n_n + 1) + no$ however this just means that the *maximum* size of the associated Cartesian program is fixed, the actual size may be anything from zero up to the maximum (in Figure 4.18 only 9 nodes are connected).

When a population of genotypes is created or mutation is applied the genes must obey certain constraints in order for the genotype to represent a valid program. These are defined as follows: Let c^n_{kj} represent the gene associated with the k-th input of a node in column j (the leftmost column is represented by 0) then it obeys the inequalities

$$
\begin{aligned}
c_{kj} &< e_{max}, j < l \\
e_{min} \leq c_{kj} &< e_{max}, j \geq l \\
e_{min} &= n_i + (j - l)n_r \\
e_{max} &= n_i + jn_r.
\end{aligned}
$$

Let c_k^o represent the gene associated with the k-th program output then it obeys the inequalities

$$h_{min} \leq c_k^o < h_{max}$$
$$h_{min} = n_i + (n_c - l)n_r$$
$$h_{max} = n_i + (n_c - 1)n_r.$$

Let c_k^f represent the gene associated with the function of k-th node then it obeys the inequality

$$0 \leq c_k^f < n_f.$$

Provided the genotypes obey these constraints, crossover can be freely applied to produce a valid solution. Modern forms of GP allow the use of Automatically Defined Functions (ADFs). In the form of CGP discussed here there are no ADFs, however as node outputs may be widely re-used one can consider it as employing Automatic Re-used Outputs (AROs). One of the attractive features of CGP is that no *explicit* encoding is required to facilitate this. A potential disadvantage is that AROs are not as general as ADFs as they can only re-use an output *with the same inputs*. One can control the amount of AROs by adjusting the levels-back parameter and the number of rows and columns. AROs are most strongly favoured in a configuration of nodes with one row, and in which $l = n_c$. At the other extreme AROs are forbidden when $n_c = 1$.

Here two evolutionary algorithms were applied. For the symbolic regression problem a generational Genetic Algorithm (GA) was used with uniform crossover, where each gene in the offspring is randomly picked up from the parents. Size two probabilistic tournament selection was used with the winner of the tournament being accepted with a probability of 0.7 (otherwise the loser was accepted). In the Santa Fe Ant Trail problem a simple form of $(l+1)$ Evolutionary Strategy (with $l \gg 4$) was used. No crossover was applied. The algorithm was as follows:

Algorithm

1. *Generate initial population at random (subject to constraints)*
2. *Evaluate fitness of genotypes in population*
3. *Promote fittest genotype to new population*
4. *Fill remaining places in the population with mutated versions of the fittest*
5. *Return to step 2 until stopping criterion reached*

Note that at step 3, if no new genotype has greater fitness but other genotypes have the same fitness as the best then one of these would be randomly chosen and promoted to the new population. This is referred to as *neutral* search. If only better genotypes are chosen then the search is referred to as *non-neutral*. For the Santa Fe trail both methods were examined.

Test Problems

Consider symbolic regression of a sixth order polynomial

$$x^6 - 2x^4 + x^2$$

The objective was to evolve a program that produces the given value of the sixth order polynomial above as its output, when the value of the one real-valued independent variable x is given as input. The program inputs were 1.0 and X. This departs from the definition given by Koza, who used *ephemeral constants* randomly defined over [−1.0, 1.0] with a given precision. Ephemeral constants are generated when the first program input is connected to a node in the initial population. From that point on in the evolutionary algorithm these constants are fixed and cannot be mutated. The function set used in this application was $\{+, -, *, div\}$, where *div* returns the numerator if the divisor is zero, otherwise it returns the normal result of division. The fitness cases were 50 random values of X from the interval $[-1.0, +1.0]$. These were fixed throughout the evolutionary run. The fitness of a program was defined as the sum over the 50 fitness cases of the sum of the absolute value of the error between the value returned by the program and that corresponding to the sixth order polynomial. The population size chosen was 10. The number of generations was equal to 8000. The crossover rate was 100% (entire population replaced by children). The mutation rate was 2%, i.e. on average 2% of the all the genes in the population were mutated. Other parameters were as follows: $n_r = 1$, $n_c = 10$, $l = 10$. An evolved genotype with a high fitness is given below (with $n_r = 4$, $n_c = 4$, $l = 4$) in Figure 4.18a:

The other problem studied was the Santa Fe Ant Trail. An imaginary ant starts at position (0,0) (top-left corner) of a 32×32 toroidal grid of squares. The ant initially faces east. There is a trail of food covering 144 squares with may gaps and twists and turns containing 89 pieces of food. The ant can move one square in the direction it is facing or turn left or right on the square that is situated on. Each of these actions require one time step. The ant has a sensor that enables it to detect whether there is food on the square one position ahead in the direction it is facing. The ant eats any food on squares that it is occupying. The objective is to evolve a program that can successfully navigate the ant so that it consumes all 89 pieces of food (success predicate). Only 600 time steps are allocated to execute the ant control program1. If a program (as is very likely) finishes before 600 time steps have elapsed then it is repeated, starting at the current state of the ant and map, this process is repeated

1 0 *3*	1 1 *3*	1 1 *2*	0 0 *2*	
4 4 *2*	2 1 *3*	3 4 *1*	1 4 *0*	
6 3 *2*	1 8 *2*	0 8 *2*	8 6 *2*	
6 11 *2*	<u>11 11 *2*</u>	5 3 *3*	13 13 *3*	15

Fig. 4.18a The genotype for a program evolved to match a sixth order polynomial. The two inputs 0 and 1 refer to 1.0 and X respectively. Functions +, −, *, div are represented by integers 0, 1, 2, 3 respectively. The program output is taken from the node with output label 15 (underlined)

until the 600 time steps have elapsed. This is a much more testing problem for Genetic Programming than the sixth order polynomial because of the time dependent behaviour of the ant. Thus one is evolving a sequential program with states that depend on past history. The function set was as used by Koza {*if-food-ahead, prog2, prog3*} coded as functions 0, 1, 2 respectively. The program inputs were coded as 0, 1, 2 and represented *move, left, right* respectively. The specific parameters used were as follows: $1 = 4$. A range of mutation rates per individual genotype were investigated from 4%–40%. Population size was 10,100 runs of 12,000 generations were carried out. Other parameters were as follows: $n_r = 1,\ n_c = 20,\ l = 20$. Thus with 81 genes and a mutation rate of 10%, 8 genes would be mutated in each genotype in the population.

Results

One method proposed for assessing the effectiveness of an algorithm, or a set of parameters was as follows. It consists of calculating the number of individual chromosomes, which would have to be processed to give a certain probability of success. To calculate this figure one must first calculate the cumulative probability of success *P(M, i)*, where *M* represents the population size, and *i* the generation number. $R(z)$ represents the number of independent runs required for a probability of success (meeting success predicate), given by z, by generation i. *I(M, z, i)* represents the minimum number of chromosomes which must be processed to give a probability of success z, by generation i. The formulae for these are given below, *Ns(i)* represents the number of successful runs at generation i, and N_{total}, represents the total number of runs

$$P(M,i) = \frac{N_s(i)}{N_{total}},\ R(z) = ceil\left\{\frac{\log(1-z)}{\log(1-P(M,i)}\right\},\ I(M,i,z) = \mathrm{M}R(z)(i+1)$$

Note that when $z = 1.0$ the formulae are invalid (all runs successful). In the expression for *I(M, i, z)*, $i + 1$ is used to taken in account the initial population. In this example, z was chosen as 0.99 so that the computational effort represented the number of evaluations required to give a probability of success of 0.99.

Sixth Order Polynomial

One hundred runs were carried out with 61 runs successful. The minimum computational effort was 90,060 that corresponded to 6 runs of 1,500 generations. It is not possible to compare this with the effort computed in as the experimental conditions were not the same and employing the fixed constant 1.0 conferred a great advantage over randomly chosen ephemeral constants.

Santa Fe Ant Trail

In Tables 4.1 and 4.2 are listed the results obtained after 100 runs for various mutation rates when the neutral and non-neutral strategies were employed. The number of solutions found with the maximum fitness is listed as #hits. The minimum computational effort I is listed in the third column (divided by 1000). The generation at which the minimum effort was observed is shown in the fourth column. Finally the number of hits that were observed at that generation is listed in the fifth column. The first thing to notice, is that some of the figures for the computational effort are calculated for a ridiculously small number of hits (less than 10, see column 5), and thus are instantly under suspicion, since in different batches of 100 runs these figures are likely to vary enormously (Actually most of figures in Table 4.2 are of this type). The computational effort is plotted against mutation rate in Figure 4.19. It would be very easy to pick out a very good figure for I and compare it favourably with results found in the literature

Table 4.1 Effort to solve the Santa Fe Ant Trail (neutral)

Mutation rate (%)	#hits	Min I/1000	Generation	#hits at generation
4	34	476	420	4
6	48	280	60	1
8	68	335	2580	30
10	83	252	2100	32
12	89	235	5880	70
14	93	173	1440	32
16	86	270	9000	79
18	85	193	2760	49
20	91	212	10,620	90
25	93	209	10,440	90
30	78	331	8280	69
40	63	511	1380	12

Table 4.2 Effort to solve the Santa Fe Ant Trail (non-neutral)

Mutation rate (%)	#hits	Min I/1000	Generation	#hits at generation
4	17	686	300	2
6	14	831	180	1
8	20	823	360	2
10	28	139	60	2
12	30	888	5220	24
14	39	458	300	3
16	42	276	120	2
18	50	204	180	4
20	48	441	900	9
25	52	324	1620	21
30	52	413	180	2
40	40	757	840	5

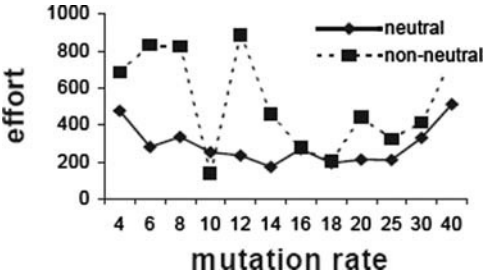


Fig. 4.19 The computational effort for neutral and non-neutral strategies for various mutation rates

(see Table 4.3). It is quite difficult to define under what conditions the calculation of computational effort is reliable without undertaking many batches of 100 runs.

To assess the quality of an algorithm one could just look at the total number of hits (second column from left in Tables 4.1 and 4.2) rather than the computational effort. This is plotted against mutation rate in Figure 4.20. Unfortunately this would give no information about the time taken by the algorithms to give these results.

It is suggested that a more reliable measure of computational efficiency might be the *hit effort* which is defined here to be the total number of evaluations over the 100 runs divided by the number of hits. The is effectively the average number of evaluations required per hit. The hit effort is most reliable when a scenario is chosen that gives many successful runs as it is fairly sensitive to the number of hits. The advantage of using hit effort is that it is a figure that is measured over the *full set of runs* and is not an inference based on assumptions about likely outcome of another experiment. Qualitatively is satisfies the requirements of a good measure of computational efficiency and quality. In the experiments performed here it presented a much more regular behaviour as can be seen in Figure 4.21 which shows the variation in hit effort with mutation rate.

Table 4.3 Previously published effort to solve the Santa Fe Ant Trail

Method	*I/1000*
Koza GP	450
Size-limited EP	136
Strict Hill Climbing	186
PDGP	336

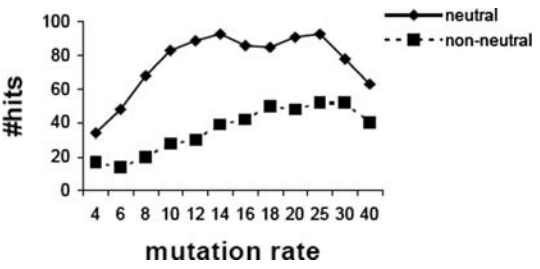


Fig. 4.20 The number of hits for neutral and non-neutral strategies for various mutation rates

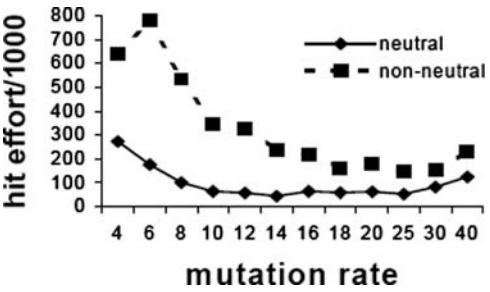


Fig. 4.21 The hit effort for neutral and non-neutral strategies for various mutation rates

It can be seen from the results presented here that high mutation rates of about 14% are most effective with neutral search, and that a higher rate of 25% are better for non-neutral search. These results support the findings that the fitness space associated with the Santa Fe trail has a great deal of randomness associated with it with the search space. It should also be noted that the maximum length of CG programs used in this application was 20, so the results may be slightly better because of this fact.

A new form of genetic programming called Cartesian Genetic Programming was discussed. It has already been shown to be very effective for Boolean function learning and here it is extended to problems with real-valued data and time dependent behaviour. The representation of genotypes in CGP has very large redundancy and it was shown that the neutrality in fitness that this allows can be used to improve the search. An important question that needs to be answered relates to which type of neutrality is most important. Other preliminary experiments for the problem of evolving binary arithmetic functions show that the search is impaired when either the node or functional redundancy is restricted when compared to a situation in which both are allowed.

4.10.3 Strongly Typed Genetic Programming (STGP)

STGP is a method of enforcing type constraints in genetic programming. Figure 4.22 illustrates some strongly typed functions along with their data types.

The changes from standard genetic programming for each genetic algorithm component is discussed in the following:

Representation

In STGP, unlike in standard genetic programming, each variable and constant has an assigned type. For example, the constants 2.1 and π have the type FLOAT, the variable V1 might have the type VECTOR-3 (indicating a three-dimensional vector), and the variable M2 might have the type MATRIX-2-2 (indicating a 2×2 matrix). Furthermore, each function has a specified type for each argument and for the value

Function Name	Arguments	Return Type
DOT-PRODUCT-3	VECTOR-3 VECTOR-3	FLOAT
VECTOR-ADD-2	VECTOR-2 VECTOR-2	VECTOR-2
MAT-VEC-MULT-4-3	MATRIX-4-3 VECTOR-3	VECTOR-4
CAR-FLOAT	LIST-OF-FLOAT	FLOAT
LENGTH-VECTOR-4	LIST-OF-VECTOR-4	INTEGER
IF-THEN-ELSE-INT	BOOLEAN INTEGER INTEGER	INTEGER

Fig. 4.22 Some strongly typed functions with their data types

it returns. In STGP, unlike in Lisp, a list must contain elements all of the same type so that the return type of CAR (and other functions returning an element of the list) can be deduced. To handle multiple data types, the definition of what constitutes a legal parse tree has a few additional criteria beyond those required for standard genetic programming, which are: (i) the root node of the tree returns a value of the type required by the problem, and (ii) each non-root node returns a value of the type required by the parent node as an argument. These criteria for legal parse trees are illustrated by the following example:

Example 1. Consider a non-terminal set N = {DOT-PRODUCT-2, DOT-PRODUCT-3, VECTOR-ADD-2, VECTOR-ADD-3, SCALAR-VEC-MULT-2, SCALAR-VEC-MULT-3} and a terminal set T = {V1, V2, V3}, where V1 and V2 are variables of type VECTOR-3 and V3 is a variable of type VECTOR-2. Let the required return type be VECTOR-3. Then, Figure 4.23 shows an example of a legal tree. Figure 4.24 shows two examples of illegal trees, the left tree because its root returns the wrong type and the right tree because in three places the argument types do not match the return types.

Evaluation Function

There are no changes to the evaluation function.

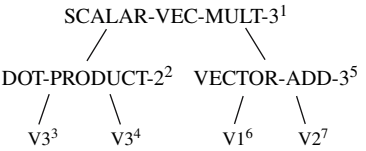


Fig. 4.23 An example of a legal tree for return type VECTOR-3

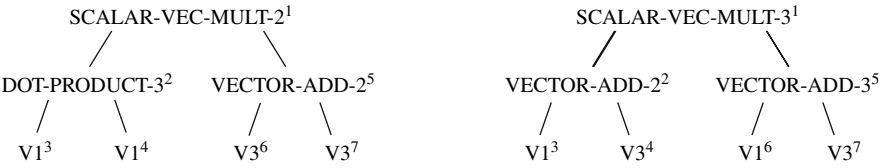


Fig. 4.24 Two examples of illegal trees for return type VECTOR-3

Initialization Procedure

The one change to the initialization procedure is that, unlike in standard genetic programming, there are type-based restrictions on which element can be chosen at each node. One restriction is that the element chosen at that node must return the expected type (which for the root node is the expected return type for the tree and for any other node is the argument type for the parent node). A second restriction is that, when recursively selecting nodes, an element which makes it impossible to select legal subtrees cannot be chosen. To implement this second restriction, it is observed that a non-terminal element can be the root of a tree of maximum depth *i* if and only if all of its argument types can be generated by trees of maximum depth *i*-1.

To check this condition efficiently, "types possibilities tables" are used, which are generated before generating the first tree. Such a table tells for each $i = 1, \ldots,$MAX-INITIAL-TREE-DEPTH what are the possible return types for a tree of maximum depth *i*. There will be two different types possibilities tables, one for trees generated by the full method and one for the grow method. Example 4 below shows that these two tables are not necessarily the same. The following is the algorithm in pseudo-code for generating these tables.

```
-- the trees of depth 1 must be a single terminal
element
loop for all elements of the terminal set
       if table_entry (1) does not yet contain this
       element's type
          then add this element's type to table_
          entry (1);
       end loop;
loop for i = 2 to MAX_INITIAL_TREE_DEPTH
-- for the grow method trees of size i−1 are also
valid trees of size i
       if using the grow method
          then add all the types from table_entry
          (i-1) to table_entry (i);
       loop for all elements of the non-terminal set
          if this element's argument types are all in
          table_entry (i−1)
```

```
            and table_entry (i) does not contain this
            element's return type
              then add this element's return type to
              table_entry (i);
            end loop;
end loop;
```

Genetic Operators

The genetic operators, like the initial tree generator, must respect the enhanced legality constraints on the parse trees. Mutation uses the same algorithm employed by the initial tree generator to create a new subtree which returns the same type as the deleted subtree and which has internal consistency between argument types and return types (Figure 4.25). If it is impossible to generate such a tree, then the mutation operator returns either the parent or nothing.

The crossover point in the first parent is still selected randomly from all the nodes in the tree. However, the crossover point in the second parent must be selected so that the subtree returns the same type as the subtree from the first parent. Hence, the crossover point is selected randomly from all nodes satisfying this constraint (Figure 4.26). If there is no such node, then the crossover operator returns either the parents or nothing.

Parameters

There are no changes to the parameters.

Generic Functions

A generic function is a function which can take a variety of different argument types and, in general, return values of a variety of different types. The only constraint is that for any particular set of argument types a generic function must return a value of a well-defined type. Specifying a set of argument types (and hence also the return type) for a generic function is called "instantiating" the generic function. Some examples of generic functions are shown in Figure 4.27. To be in a parse tree,

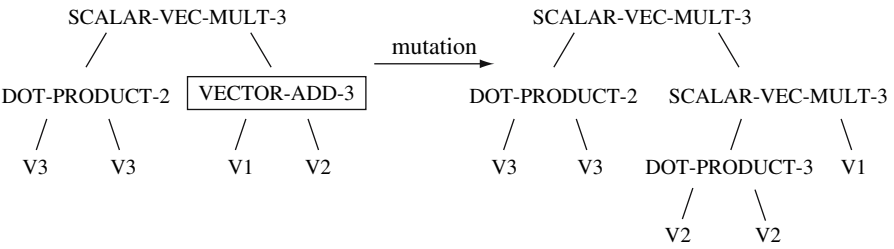


Fig. 4.25 Mutation for STGP

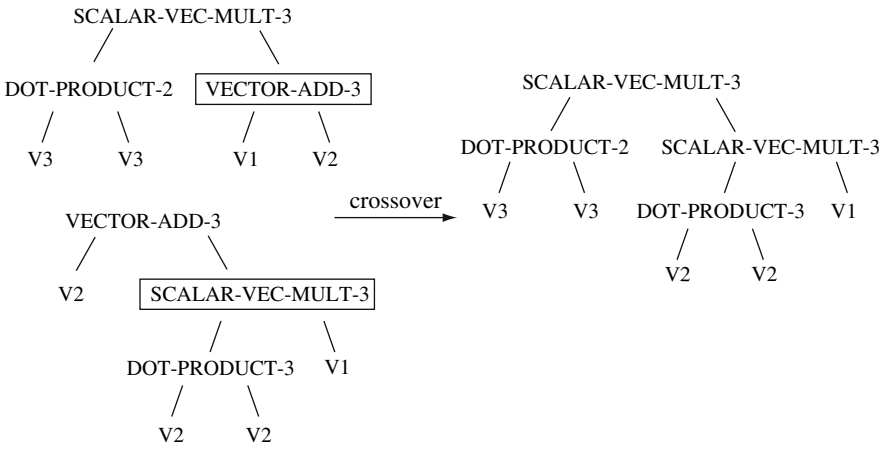


Fig. 4.26 Crossover for STGP

a generic function must be instantiated. Once instantiated, an instance of a generic function keeps the same argument types even when passed from parent to child. Hence, an instantiated generic function acts exactly like a standard strongly typed function. A generic function gets instantiated during the process of generating parse trees (for either initialization or mutation). There can be multiple instantiations of a generic function in a single parse tree. Because generic functions act like standard strongly typed functions once instantiated, the only changes to the STGP algorithm needed to accomodate generic functions are for the tree generation procedure. There are three such changes required.

First, during the process of generating the types possibilities tables, recall that for standard non-terminal functions it is required only to check that each of its argument

Function Name	Arguments	Return Type
DOT-PRODUCT	VECTOR-i VECTOR-i	FLOAT
VECTOR-ADD	VECTOR-i VECTOR-i	VECTOR-i
MAT-VEC-MULT	MATRIX-i-j VECTOR-j	VECTOR-i
CAR	LIST-OF-t	t
LENGTH	LIST-OF-t	INTEGER
IF-THEN-ELSE	BOOLEAN t t	t

Fig. 4.27 Some generic functions with their argument types and return types. Here, i and j are arbitrary integers and t is an arbitrary data type

types was in the table entry for depth i-1 in order to add its return type to the table entry for depth i. This does not work for generic functions because each generic function has a variety of different argument types and return types. For generic functions, this step is replaced with the following:

```
loop over all ways to combine the types from
table_ entry (i−1) into
  sets of argument types for the function
       if the set of arguments types is legal
       and the return type for this set of argument
       types is not in table_entry (i)
            then add the return type to table_entry
            (i);
       end loop;
```

The second change is during the tree generation process. For standard functions, when deciding whether a particular function could be child to an existing node, independent check can be performed to find whether it returns the right type and whether its argument types can be generated. However, for generic functions, these two tests must be replaced with the following single test:

```
loop over all ways to combine the types from table_
entry (i-1)  into
  sets of argument types for the function
       if the set of arguments types is legal
       and the return type for this set of argument
       types is correct then return that this function is
       legal;
       end loop;
       return that this function is not legal;
```

The third change is also for the tree generation process. Note that there are two types of generic functions, ones whose argument types are fully determined by selection of their return types and ones whose argument types are not fully determined by their return types. The latter are known as “generic functions with free arguments”. Some examples of generic functions with free arguments are DOT-PRODUCT and MAT-VEC-MULT, while some examples of generic functions without free arguments are VECTOR-ADD and SCALAR-VEC-MULT. When a generic function with free arguments is selected to be a node in a tree, its return type is determined by its parent node (or if it is at the root position, by the required tree type), but this does not fully specify its argument types. Therefore, to determine its arguments types and hence the return types of its children nodes, the types possibilities table should be used to determine all the possible sets of argument types which give rise to the determined return type (there must be at least one such set for this function to have been selected) and randomly select one of these sets.

(a)

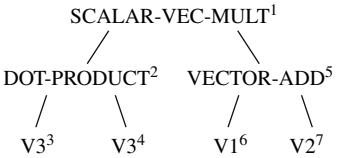


Fig. 4.27a A legal tree using generic functions

Example: Using generic functions, the non-terminal set from Example 1 can be rewritten in a more compact form: N = {DOT-PRODUCT, VECTOR-ADD, SCALAR-VEC-MULT}. Recall that T = {V1, V2, V3], where V1 and V2 are type VECTOR-3, and V3 is type VECTOR-2. Figure 4.27a shows the equivalent of the tree in Figure 4.23. To generate the tree shown in Figure 4.27a as an example of a full tree of depth 3, the following steps are performed.

Step 1: At point 1, either VECTOR-ADD or SCALAR-VEC-MULT can be selected, and SCALAR-VEC-MULT is choosen.
Step 2: At point 2, DOT-PRODUCT is selected. Since DOT-PRODUCT has free arguments, its argument types are selected.
Step 3: Then, points 3 and 4 must be of type VECTOR-2 and hence must be V3.
Step 4: Point 5 must be VECTOR-ADD.
Step 5: Points 6 and 7 can both be either V1 or V2, and thus V1 is chosen for point 6 and V2 is chosen for point 7.

Generic Data Types

A generic data type is not a true data type but rather a set of possible data types. Examples of generic data types are VECTOR-GENNUM1, which represents a vector of arbitrary dimension, and GENTYPE2, which represents an arbitrary type. Generic data types are treated differently during tree generation than during tree execution. When generating new parse trees (either while initializing the population or during reproduction), the quantities such as GENNUM1 and GENTYPE2 are treated like algebraic quantities. Examples of how generic data types are manipulated during tree generation are shown in Figure 4.28.

During execution, the quantities such as GENNUM1 and GENTYPE2 are given specific values based on the data used for evaluation. For example, if in the evaluation data, a particular vector of type VECTOR-GENNUM1 is chosen to be a two-dimensional vector, then GENNUM1 is equal to two for the purpose of executing on this data. The following two examples illustrate generic data types. An advantage of using generic data types is that, when using generic data types, the functions that are learned during genetic programming are generic functions.

Function	Arguments	Return Type
VECTOR-ADD	VECTOR-GENNUM2 VECTOR-GENNUM2	VECTOR-GENNUM2
VECTOR-ADD	VECTOR-GENNUM2 VECTOR-3	illegal different dimensions
VECTOR-ADD	VECTOR-GENNUM2 VECTOR-GENNUM1	illegal different dimensions
VECTOR-ADD	GENTYPE1 GENTYPE1	illegal not vectors
CAR	LIST-OF-GENTYPE3	GENTYPE3
CAR	GENTYPE3	illegal, not a list
IF-THEN-ELSE	BOOLEAN GENTYPE3 GENTYPE3	GENTYPE3

Fig. 4.28 Some examples of how generic data types are manipulated

STGP's Programming Language

The programming language for STGP is a cross between Ada and Lisp. The essential ingredient it takes from Ada is the concept of strong typing and the concept of generics as a way of making strongly typed data and functions practical to use. The essential ingredient it takes from Lisp is the concept of having programs basically be their parse trees. The resulting language might best be considered a strongly typed Lisp. There are reasons why a strongly typed Lisp is a good language not only for learning programs using genetic algorithms but also for any type of automatic learning of programs. Having the programs be isomorphic to their parse trees makes the programs easy to create, revise and recombine. This not only makes it easy to define genetic operators to generate new programs but also to define other methods of generating programs. Strong typing makes it easy to ensure that the automatically generated programs are actually type-consistent programs.

Conclusion

Thus the concept of Strongly Typed Genetic Programming (STGP) was discussed. STGP is an extension to genetic programming which ensures that all parse trees which are generated obey any constraints on data types. By doing this, STGP can greatly decrease the search time and/or greatly improve the generalization performance of the solutions it finds. The concepts of generic functions, generic data types and programming language were discussed as a way of making STGP more practical, more powerful, and more oriented towards code reuse.

4.11 Advantages of using Genetic Programming

A few advantages of genetic programming are:

- No analytical knowledge is needed and still accurate results are obtained.
- If fuzzy sets are encoded in the genotype, new and more suited- fuzzy sets are generated to describe precise and individual membership functions. This can be done by means of the intersection and/or union of the existing fuzzy sets.
- Every component of the resulting GP rule-base is relevant in some way for the solution of the problem. Thus null operations that will expend computational resources at runtime are not encoded.
- This approach does scale with the problem size. Some other approaches to the cart centering problem use a GA that encodes NxN matrices of parameters. These solutions work bad as the problem grows in size (i.e. as N increases).
- With GP no restrictions are imposed on how the structure of solutions should be. Also the complexity or the number of rules of the computed solution is not bounded.

Summary

From the sections discussed in this chapter it is inferred that Genetic programming (GP) is an automated method for creating a working computer program from a high-level problem statement of a problem. Genetic programming starts from a high-level statement of "what needs to be done" and automatically creates a computer program to solve the problem. The fact that genetic programming can evolve entities that are competitive with human-produced results suggests that genetic programming can be used as an automated invention machine to create new and useful patentable inventions. In acting as an invention machine, evolutionary methods, such as genetic programming, have the advantage of not being encumbered by preconceptions that limit human problem-solving to well-troden paths.

In this chapter, an introduction and definition of GP along with a brief history was discussed. To get an idea about programming a basic introduction to Lisp Programming Language was dealt. The basic operations of GP were discussed along with an illustration. The steps of GP were also illustrated along with a flow chart and enhanced versions of GP such as Meta GP, Cartesian GP and Strongly Typed GP were also elaborated in this chapter.

Genetic programming has delivered a progression of qualitatively more substantial results in synchrony with five approximately order-of-magnitude increases in the expenditure of computer time since 1987.

Review Questions

1. Define Genetic Programming.
2. Write a short note on Lisp programming language.
3. Explain the basic operations of Genetic Programming.
4. Explain selection in Genetic Programming.
5. How does crossover and mutation occur in Genetic Programming.
6. Mention the paradigms of GA in machine learning and give one example for each.
7. What are the five basic preparatory steps that should be specified by the user in Genetic Programming?
8. Draw and explain the flowchart of Genetic Programming.
9. Mention the enhanced versions of Genetic Programming.
10. Explain the basic technique of operation of Meta Genetic Programming.
11. What is the methodology behind Cartesian Genetic Programming
12. Explain the Cartesian Genetic Programming Algorithm.
13. Describe an example of Cartesian Genetic Programming in detail.
14. How does Strongly Typed Genetic Programming differ from standard Genetic Programming?
15. Write a short note on genetic operators and genetic functions in Strongly Typed Genetic Programming?
16. Mention a few advantages of Genetic Programming.

Chapter 5
Parallel Genetic Algorithms

Learning Objectives: On completion of this chapter the reader will have complete knowledge on:

- An Overview of Parallel and Distributed Computer Architectures
- Classification of PGA
- Population Models
- Models Based on Distribution of Population
- PGA Models based on Implementation
- PGA Models based on Parallelism
- Communication Topologies
- Hierarchical PGA
- Advantages of PGA

5.1 Introduction

Genetic Algorithms (GAs) are efficient search methods based on principles of natural selection and genetics. They are being applied successfully to find acceptable solutions to problems in business, engineering, and science. GAs are generally able to find good solutions in reasonable amounts of time, but as they are applied to harder and bigger problems there is an increase in the time required to find adequate solutions. As a consequence, there have been multiple efforts to make GAs faster, and one of the most promising choices is to use parallel implementations. But making a GA faster is not the only advantage that can be expected when designing a parallel GA. A Parallel Genetic Algorithm (PGA) has an improved power to tackle more complex problems since it can use more memory and CPU resources. The versatility is larger than in a sequential version since different solution species can evolve and coexist in parallel, thus yielding alternative solutions to the same problem. An additional advantage is that the string diversity is better sustained when run in parallel, which is good for tackling new domains such as non-stationary problems where the optimum varies and moves during the search. Finally, parallel GAs

reduces the probability of getting stuck in a local optimum (efficacy) and they provide a clear way to interact in parallel with other optimization algorithms (enhanced applicability).

Parallel GAs are a class of guided random evolutionary algorithms. The most popular parallel GAs consists in multiple populations that evolve separately most of the time and exchange individuals occasionally. If natural evolution is mimicked it would not operate on a single population in which a given individual has the potential to mate with any other partner in the entire population (*panmixia*). Instead, species would tend to reproduce within *subgroups* or within *neighborhoods*. A large population distributed among a number of semi-isolated breeding groups is known as *polytypic*. This method divides the population into subpopulations called demes. Demes are separated from one another (geographic isolation) and individuals compete only within a deme. An additional operator called migration is used to move the individuals from one deme to another. If the individuals can migrate to any other deme, the model is called an island model.

The local selection and reproduction rules allow the species to evolve locally, and diversity is enhanced by migrations of strings among demes.

This type of parallel GAs is called multi-deme, coarse-grained or distributed GAs, and this chapter concentrates on this class of algorithm. However, this chapter also describes the other major types of parallel GAs with some examples.

5.2 Parallel and Distributed Computer Architectures: An Overview

Parallelism can arise at a number of levels in computer systems: task level, instruction level or at some lower machine level. Although there are several ways in which parallel architectures may be classified, the standard model of Flynn is widely accepted as a starting point. But the reader should be aware that it is a coarse-grain classification: for instance, even today's serial processors are in fact highly parallel in the way in which they execute instructions, as well as with respect to the memory interface. Even at a higher architectural level, many parallel architectures are in fact hybrids of the base classes.

Flynn's taxonomy is based on the notion of instruction and data streams. There are four possible combinations conventionally called SISD (single instruction, single data stream), SIMD (single instruction, multiple data stream), MISD (multiple instructions, single data stream) and MIMD (multiple instruction, multiple data stream). Figure 5.1 schematically depicts the three most important model architectures.

The SISD architecture corresponds to the classical mono-processor machine such as the typical PC or workstation. As stated above, there is already a great deal of parallelism in this architecture at the instruction level (pipelining, superscalar and very long instruction execution). This kind of parallelism is "invisible" to the

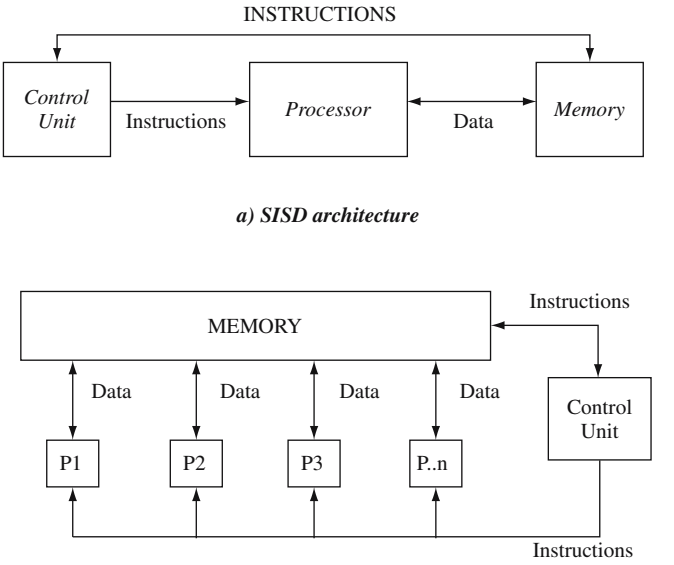


Fig. 5.1 Flynn's Model of Parallel architectures

programmer in the sense that it is built-in in the hardware or must be exploited by the compiler.

In the SIMD architecture the same instruction is broadcast to all processors. The processors operate in lockstep executing the given instruction on different data stored in their local memories (hence the name: single instruction, multiple data). The processor can also remain idle, if this is appropriate. SIMD machines exploit spatial parallelism that may be present in a given problem and are suitable for large, regular data structures. If the problem domain is spatially or temporally irregular, many processors must remain idle at a given time step since different operations are called for in different regions. Obviously, this entails a serious loss in the amount of

parallelism that can be exploited. Another type of SISD computer is the vector processor, which is specialized, in the pipelined execution of vector operations. This is the typical component of supercomputers but it is not a particularly useful architecture for evolutionary algorithms. In the MIMD class of parallel architectures multiple processors work together through some form of interconnection. Different programs and data can be loaded into different processors, which mean that each processor can execute different instructions at any given point in time. Usually the processors will require some form of synchronization and communication in order to cooperate on a given application. This class is the more generally useful and most commercial parallel and distributed architectures belong to it.

There has been little interest up to now in the MISD class since it does not lend itself to readily exploitable programming constructs. One type of architecture of this class that enjoys some popularity is the so-called systolic arrays, which are used, in specialized applications such as signal processing.

Another important design decision is whether the system memory spans a single address space or it is distributed into separated chunks that are addressed independently. The first type is called shared memory while the latter is known as distributed memory. This is only a logical subdivision independent of how the memory is physically built.

In shared memory multiprocessors all the data are accessible by all the processors. This poses some design problems for data integrity and for efficiency. Fast cache memories next to the processors are used in order to speedup memory access to often-used items. Cache coherency protocols are then needed to insure that all processors see the same value for a given piece of data.

Distributed memory multicomputers is a popular architecture which is well suited to most parallel workloads. Since the address spaces of each processor are separate, communication between processors must be implemented through some form of message passing. To this class belong networked computers, sometimes called computer clusters. This kind of architecture is interesting for several reasons. First of all, it has a low cost since already existing local networks of workstations can be used just by adding a layer of communication software to implement message passing. Second, the machines in these networks usually feature up-to-date off-the-shelf processor and standard software environments, which make program development easier. The drawbacks are that parallel computing performance is limited by comparatively high communication latencies and by the fact that the machines have different workloads at any given time and are possibly heterogeneous. Nevertheless, problems that do not need frequent communication are suitable for this architecture. Moreover, some of the drawbacks can be overcome by using networked computers in dedicated mode with a high-performance communication switch. Although this solution is more expensive, it can be cost-effective with respect to specialized parallel machines. Also web technology can be used for parallel computing and for both computing and information related applications. Harnessing the Web or some other geographically distributed computer resources so that it looks like a single computer to the user is called metacomputing. The concept is very attractive but many challenges remain. In fact, in order to transparently and efficiently

distribute a given computational problem over the available resources without the user taking notice requires advances in the field of user interfaces and in standardized languages, monitoring tools and protocols to cope with the problem of computer heterogeneity and uneven network load. The Java environment is an important step in this direction.

5.3 Classification of PGA

There are two main reasons for parallelizing genetic algorithm: one is to achieve time savings by distributing the computational effort and the second is to benefit from a parallel setting from the algorithmic point of view, in analogy with the natural parallel evolution of spatially distributed populations.

The first type of parallel genetic algorithm makes use of the available processors or machines to run independent problems. This is trivial, as there is no communication between the different processes and for this reason it is sometimes called an embarrassingly parallel algorithm. This extremely simple method of doing simultaneous work can be very useful. For example, this setting can be used to run several versions of the same problem with different initial conditions, thus allowing gathering statistics on the problem. Since genetic algorithms are stochastic in nature, being able to collect this kind of statistics is very important. This method is in general to be preferred with respect to a very long single run since improvements are more difficult at later stages of the simulated evolution. Other ways in which the model can be used is to solve N different versions of the same problem or to run N copies of the same problem but with different GA parameters, such as crossover or mutation rates. Neither of the above adds anything new to the nature of the genetic algorithms but the timesavings can be large.

In genuine parallel genetic algorithms models, there are several possible levels at which an evolutionary algorithm can be parallelized: the population level, the individual level or the fitness evaluation level. Moreover, although genetic algorithms and genetic programming are similar in many respects, the differences in the individual representation make genetic programming a little bit different when implemented in parallel.

The basic idea behind most parallel programs is to divide a task into subtasks and to solve the subtasks simultaneously using multiple processors. This divide-and-conquer approach can be applied to GAs in many different ways, and the literature contains many examples of successful parallel implementations. Some parallelization methods use a single population, while others divide the population into several relatively isolated subpopulations. Some methods can exploit massively parallel computer architectures, while others are better suited to multicomputers with fewer and more powerful processing elements.

Figure 5.2 shows the classification of parallel GAs according to the parallel model, the distribution of the population and the implementation. This is not the

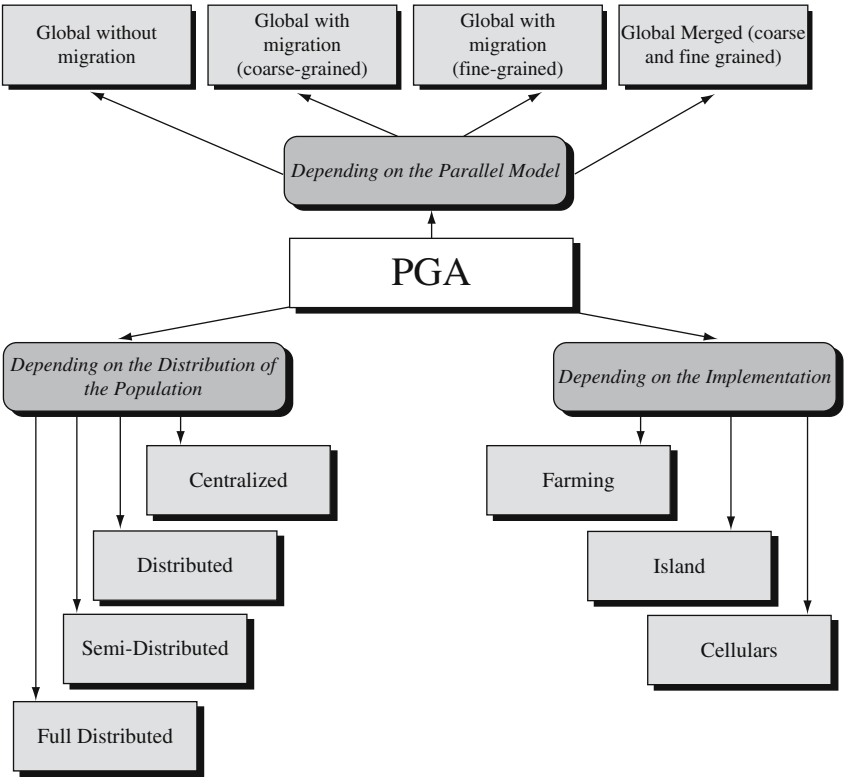


Fig. 5.2 Classification of Parallel GAs

sole classification one can make for PGAs because it is very normal that the existing PGAs combine several characteristics from these apparently disjoin models.

5.4 Parallel Population Models for Genetic Algorithms

Since genetic algorithms are a whole class of algorithms, their parallelization is not only a functionally equivalent transformation of a given sequential algorithm into a parallel implementation. More than that, from the set of genetic algorithm strategies, such strategies have to be selected and their parameters have to be adjusted so that the following parallelization yields an efficient implementation. Moreover it is possible to design new and easier parallelizable variants of the genetic algorithm strategies. The parallelization of genetic algorithms is thus preceded by the modification of the structure of the genetic algorithm into an easier parallelizable form.

Such a modified structure of the genetic algorithm is also called as its parallel population model. This model describes basically, how the population is divided

into subpopulations and how the information exchange between these subpopulations is organized. Thus the parallel population model basically mirrors the memory and communication structure of a parallel computer. While the parallel computers for some early parallel implementations of the genetic algorithm are not available any more, their parallel population model is still interesting. These models could also be used for the implementation of genetic algorithms in several modern parallel computers. On the other hand the structural modifications introduced by the parallel population models also have an influence on the effectiveness of the genetic search. If this influence is negative, the question raises how much parallelism is additionally needed to compensate. If the influence is positive, then the parallel population models should also be discussed in the non-parallel context.

5.4.1 Classification of Global Population Models

A suitable classification scheme should clearly express how the genetic algorithms differ structurally with respect to the classical genetic algorithm. These structural differences are mainly concerned with differences in the selection operator (i.e. mating selection and survival selection), since reproduction (i.e. crossing-over and mutation) and evaluation are not affected by parallelization. The classical genetic algorithm is characterized by the fact that selection is global and any pairing of the individuals is possible. Other parallel population models are characterized by different approaches to a non-global selection. It is thus proposed to classify the parallel population models with respect to the radius of their selection operator:

Global Model: The population is unstructured. Selection is global, i.e. the fitness of any individual is calculated relative to all other individuals and principally any individuals could be selected as reproduction partners.
Regional Model: The population is divided into subpopulations (regions), the fitness of individuals is calculated relative to individuals of the same region and the reproduction partner comes preferably from the same region.
Local Model: The population is provided with a neighborhood structure, the fitness of individuals is calculated relative to those in the local neighborhood and the reproduction partner comes preferably from the same neighborhood.

The relation between these three models can be illuminated from different perspectives: From the perspective of the population, the regional and the local model are two possibilities to divide the population into smaller units. In this view the local model with its small neighborhoods is the natural opponent to the global model and the regional model is just an intermediate model with larger neighborhoods. From the perspective of the individual, the models define different limitations for the interactions between individuals. In this view the regional model with its separated regions is the natural opponent to the global model and the local model is just an intermediate model with overlapping regions.

5.4.2 Global Population Models

Global models implement the classical genetic algorithm. The fitness-proportional mating selection is a serial bottleneck in the classical genetic algorithm. After pairs of individuals are selected, they also have to be moved to the same processor. Depending from the mapping of the population into the storage of a parallel processor, arbitrary conflicts can occur and arbitrary communication patterns between processors have to be realized.

5.4.2.1 Global Selection Strategies

Global population models differ mainly whether and how the problems with the parallelization of the global selection are solved.

Models with sequential selection: The easiest possibility is to omit the parallelization of the global selection. In this case typically a synchronous or asynchronous master-slave programming model is used. The population resides completely in the memory of a master process. Such a parallelization can be advocated if an extremely expensive objective function is given for the considered optimization problem and the computing time for selection can thus be neglected. If the population is stored in the memory of a single master process, it can be advantageous to do reproduction also sequentially, because then instead of a pair of parents only one offspring has to be transmitted to the slave processes.

Models with parallel selection: The calculation of the absolute fitness for fitness-proportional mating selection can be parallelized by parallelizing the necessary summation. In the best case this will result in a logarithmic effort. For rank-proportional mating selection, the population has to be sorted, which can also be done with logarithmic effort on a parallel computer, e.g. by bitonic sort. The tournament mating selection, where a limited number of randomly selected individuals are to be compared is easier to parallelize. The tournament mating selection thus seems to be best suited for parallel implementations of the global population model. The more difficult problem, to bring the mating individuals together without memory conflicts or communication bottlenecks, remains.

5.4.3 Regional Population Models

Regional models are an extension to the structure of the classical genetic algorithm. The population is first divided into subpopulations (or regions, or demes). Each region runs a separate genetic algorithm. The global selection is replaced by a ‘regional selection’, i.e. the fitness of an individual is calculated only relative to other individuals from the same region, the reproduction partners come from the same

region and the offspring compete with their parents for space in the same region. Additionally to the basic operators of a genetic algorithm an operator 'migration' is introduced, which controls the exchange of individuals between the regions. Several strategies and parameters determine how this is done in detail:

5.4.3.1 Regional Structures

By dividing the population into regions and by specifying a migration topology, the regional model can be adapted to a given parallel computer architecture. The structure of a regional model thus reflects the structure of the parallel computer and can only be varied in narrow limits. Recently the regional model has also been implemented on sequential computers, thus allowing consideration of arbitrary structures.

Number of regions: In most cases the number of regions remains fixed and is equal to the number of available processors. Typical values are powers of two from $R = 4$ to $R = 64$. With a constant size of the total population, the number of regions also determines the number of individuals S in each region. Typically each region contains a subpopulation with $S = 50$ up to $S = 100$ individuals.

Migration topology: The migration topology determines the migration paths along which individuals can move to other regions. In most cases the topology is similar to the interconnection topology of a given parallel processor, for example a hypercube or a two-dimensional grid. If the number of regions is small, then a full migration topology can be considered, where all regions are connected with all other regions. In this case a migrating individual is present in all other regions at once.

Some studies of the regional model follow the assumption that the partial isolation of the regions is advantageous for the optimization process and a very fast distribution of good individuals is not recommended. For example, in hypercube topology, in each migration phase only the migration along one dimension of the hypercube is allowed, as this topology has a very small diameter. In a ring topology, the i-th migration phase region r exchanges individuals with region $\mathrm{r} + i \bmod R$ .In the 'Ladder' topology R regions are connected to a toroidal $2 \times \mathrm{r}/2$grid, so that the topology has a substantially larger diameter than a square grid.

Magnitude of Migration: The exchange of individuals between the regions is controlled by the two parameters 'migration rate' and 'migration interval'. A large magnitude of migration lets regional models work similar to the global model, while a small magnitude of migration leads to several independent runs of the genetic algorithm with smaller populations.

Migration rate: During each migration phase, one or several individuals can migrate. The migration rate μ is given as absolute number or in percent of the subpopulation size. A migration rate larger than one is sometimes recommended, because a single individual possibly has a smaller chance to survive in the other region if

it is recombined with completely different individuals. Typical migration rates are between 10 and 20% of the subpopulation size.

Migration interval: A migration phase can be included either in each iteration of the genetic algorithm or only after a certain interval of iterations (epochs). A larger migration interval is normally used in connection with a larger migration rate. An alternative is that the migration should occur precisely when the genetic algorithm has reached (premature) convergence on the regions. Also the migration can be performed asynchronously each time an improved individual is found in a region.

Migration strategies: Basically migration can use the same strategies as they are used for mating selection and survival selection. Since regional selection already favour better individuals inside regions, migration could also use other criteria for selection and replacement.

Migration selection: In most cases the best or a limited sequence of the best individuals are selected for migration. It is also possible to choose a fitness-proportional or rank-proportional selection scheme as it is used for mating selection. Or in each migration phase several individuals are selected and replaced randomly.

Migration replacement: In most cases only the worst or a limited sequence of the worst individuals are replaced. Similar to the migration selection, a fitness-proportional or rank-proportional selection scheme can be used under opposite sign. Again this would impose a selection pressure towards better individuals inside the region. Other strategies for migration replacements were designed with the goal to maintain a large variability inside the regions. In one method the most similar individuals measured by hamming distance of the genotypes-to the immigrants are replaced. In the 'emigration model', the immigrating individuals replace the emigrants of this region.

5.4.4 Local Population Models

Local population models are a modification—rather than an extension—of the structure of genetic algorithms. The population is provided with a spatial structure. On the whole population runs a genetic algorithm, whereby the global selection is replaced by a 'local selection'. The fitness of an individual is computed only relative to individuals of a local neighborhood, reproduction partners come preferably from the same neighborhood and the offspring competes with its parents for space in the same neighborhood. Thus basically the selection operator is modified. Local population models can be described by the following strategies and parameters:

5.4.4.1 Neighborhood Structures

The neighborhood structures determine the spatial placement of individuals of a population and lay down the paths along which individuals can propagate through the population. They were also primarily introduced to adapt genetic algorithms to the structure of memory and communication of a given parallel computer.

Neighborhood topology: The neighborhood topology τ_N determines which individuals belong to the local neighborhood of an individual. For very small populations similar arguments as for the selection of a migration topology hold and a topology with large diameter should be used. For larger populations a square toroidal grid with four or eight neighbors is used frequently.

Neighborhood radius: Basically only the direct neighbors of a topology belong to the local neighborhood. The neighborhood radius ρ_N allows to size up the neighborhood by including all individuals that are reached in a fixed number of steps.

5.4.4.2 Local Selection Strategies

For the local model the global selection is replaced by a local selection. Basically the same selection strategies could be used for the local model, but it has to be considered that with the reach of the selection also the selection pressure decreases. The selection pressure is thus additionally dependent of the neighborhood structure.

Local mating selection: For all positions of the neighborhood topology a pair of individuals is determined by local mating selection. There is a distinction whether both mating partners are selected from the local neighborhood, or if only one individual is selected from the local neighborhood as mating partner for the individual sitting in the centre of the neighborhood. Similar to global mating selection, a local fitness-proportional or rank-proportional selection as well as a tournament selection can be considered. A further local mating selection strategy, the random walk mating, was investigated by COLLINS and JEFFERSON (1991). For each position of the neighborhood topology both parents are selected as the best individual along a randomly generated path.

Local survival selection: By local survival selection is determined, which individuals from the local neighborhood are to be replaced by the offspring. Generational replacement can be used without modifications as local survival selection strategy if the offspring automatically replaces the individual in the centre of a neighborhood. Elitist replacement cannot be transferred to local survival selection since global knowledge about the whole population would be necessary.

5.5 Models Based on Distribution of Population

Depending on the distribution of the population PGA's are classified into

Centralized PGA
Distributed PGA

The two models are discussed in the following section.

5.5.1 Centralized PGA

The centralized also known as compact genetic algorithm (cGA) uses an encoding directly connected with the solution of the problem. The code is based on the edges, which belong to the spanning tree. A number is assigned to every edge of the tree after partitioning. Consequently a chromosome will have k-1 genes for a k-way partitioning, and the value of these genes can be any of the order values of the edges.

For example, in the graph represented in Figure 5.3 a, a chromosome could be (3 4 6) for a 4-way partitioning. This means that this solution is obtained after the suppression of edge numbers 3, 4, and 6 from the spanning tree. So the alphabet of the algorithm is: $\Omega = \{0, 1, \ldots, n-1\}$ where n is the number of vertices of the target graph (circuit), because the spanning tree has n-1 edges as shown in Figure 5.3 b. The figure shows the meaning of the chromosome 3 4 6.

Base algorithm

The cGA represents the population by means of a vector of values *pi* < @1;>*[0,1], for* i=*1..ℓ*, where ℓ is the number of alleles needed to represent the solutions, and it is also called probability vector. Each value *pi* measures the proportion of individuals in the simulated population, which have a zero (one) in the *ith* locus of their representation. By treating these values as probabilities, new individuals can be generated and, based on their fitness, the probability vector updated accordingly to favor the generation of better individuals. The values for probabilities *pi* are initially set to *0.5* to represent a randomly generated population in which the

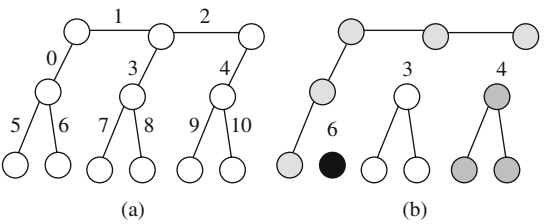


Fig. 5.3 An example code

value each allele has equal probability. In each iteration, the cGA generates two individuals on the basis of the current probability vector and compares their fitness. Let $W = [W_0, W_1, , W_2, , \ldots . W_\ell,]$, be the representation of the individual with better fitness and $L = [L_0, L_1, L_2, \ldots .. L_\ell,]$ be one of the individual whose fitness was worse. The competitor representations are used to update the probability vector at step $k + 1$ in the following way:

$$p_i^{k+1} = \begin{cases} p_i^k + 1/n \quad if \quad W_i = 1 \quad or \quad L_i = 0 \\ p_i^k + 1/n \quad if \quad W_i = 0 \quad or \quad L_i = 1 \\ \qquad\qquad p_i^k \quad if \quad W_i = L_i \end{cases} \tag{5.1}$$

Where n is the dimension of the population simulated, and *Wi (Li)* is the value of the ith allele of W (L). The cGA ends when the values of the probability vector are all equal to *0* or *1* (this process will take a different number of iteration depending on the complexity of the problem and the size of the population). At this point the vector p itself represents the final solution. In order to represent a given population of n individuals, the cGA updates the probability vector by a constant value equal to $1/n$. Larger population dimension can be exploited without significantly increasing memory requirements, but only slowing cGA convergence. This peculiarity makes the use of cGAs very attractive to solve problems for which the huge memory requirements of GAs are a constraint. To solve higher than order-one problems GAs with both higher selection rates and larger population sizes have to be exploited. The cGA selection pressure (Ps) can be increased by modifying the algorithm in the following way:

1. At each iteration generate 's' individuals instead of two from the probability vector;
2. Choose among the 's' individuals the one with best fitness and select as W its representation
3. Compare W with the other 's-1' representations and update the probability vector accordingly and in the same way W and L are compared.

The other parts of the algorithm remain unchanged. Such an increase on the selection pressure helps the cGA to converge to better solutions since it increases the survival probability of higher order building blocks. To conclude cGA overcomes scalability problem.

5.5.2 Distributed PGA

When dealing with a distributed GA some additional parameters need to be defined. If migration of individuals has to take place then the rate for the migrations and the number of individuals to migrate come into scene. A decision has to be made on which individuals are going to migrate. Finally the decision on the target subpopulation determines the global topology of the distributed system. Traditionally,

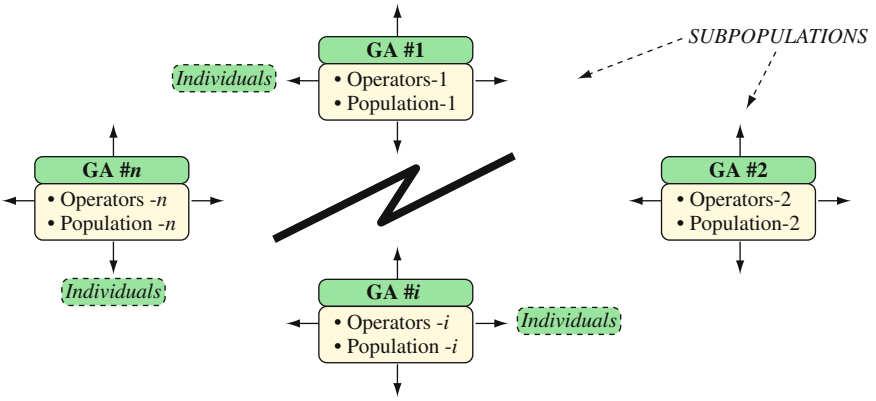


Fig. 5.4 A Distributed (Migration) GA

the topology is a ring, one individual - the best or a random individual - migrates in every migration step and migrations take place at some predefined and periodic moments (although some criteria for more controlled migrations have been considered in this field, usually performing migrations when some condition holds -the standard deviation falls below a given bottom limit, etc…-). Therefore distributed GA is also known as Migration GA whose schematic is illustrated in Figure 5.4.

The migration scheme is very much suitable for implementation in a network of workstations. The existing UNIX networks with TCP/IP communication protocols are the preferred for many researchers due to its relatively easy used and also to the existence of many communication tools: socket interface, MPI, PVM, etc.

5.6 PGA Models Based on Implementation

Depending on the Implementation PGAs are classified into Master –Slave PGA, Island PGA and Cellular PGA.

5.6.1 Master–slave/Farming PGA

In the beginning of the parallelization of these algorithms the well-known master-slave farming PGA was used. In this method there is a single panmictic population (just as in a simple GA), but the evaluation of fitness is distributed among several processors as shown in Figure 5.5. The central processor performs the selection operations while the associated slave processors perform the recombination, mutation and the evaluation of the fitness function. The last operation (evaluation) usually

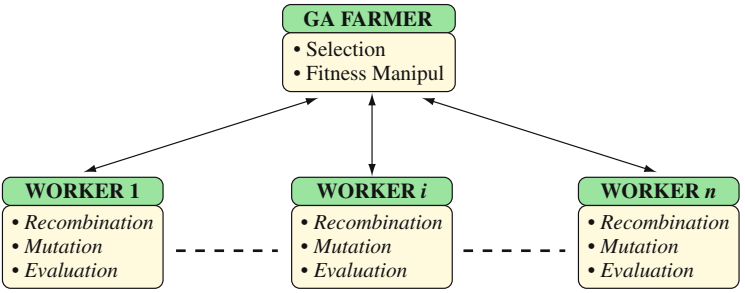


Fig. 5.5 A Farmed GA

represents the most time-consuming load of the genetic algorithm. Since in this type of parallel GA, selection and crossover consider the entire population it is also known as global parallel GAs. As in the serial GA, each individual may compete and mate with any other (thus selection and mating are global). Global parallel GAs are usually implemented as master-slave programs, where the master stores the population and the slaves evaluate the fitness.

The most common operation that is parallelized is the evaluation of the individuals, because the fitness of an individual is independent from the rest of the population, and there is no need to communicate during this phase. The evaluation of individuals is parallelized by assigning a fraction of the population to each of the processors available. Communication occurs only as each slave receives its subset of individuals to evaluate and when the slaves return the fitness values.

If the algorithm stops and waits to receive the fitness values for all the population before proceeding into the next generation, then the algorithm is synchronous. A synchronous master-slave GA has exactly the same properties as a simple GA, with speed being the only difference. Synchronous GPGA has the same properties as sequential genetic algorithm, but it is faster if the algorithm spends most of the time for the evaluation process. Synchronous master-slave GA has many advantages: they explore the search space exactly as a sequential GA, they are easy to implement and significant speedup is possible if the communication costs does not dominate the computation costs. But the bottleneck effect is that the whole process has to wait for the slowest processor to finish its fitness evaluations. After that, the selection operators can be applied. However, it is also possible to implement an asynchronous master-slave GA where the algorithm does not stop to wait for any slow processors, but it does not work exactly like a simple GA.

The global parallelization model does not assume anything about the underlying computer architecture, and it can be implemented efficiently on shared-memory and distributed-memory computers. On a shared-memory multiprocessor, the population could be stored in shared memory and each processor can read the individuals assigned to it and write the evaluation results back without any conflicts.

On a distributed-memory computer, the population can be stored in one processor. This “master” processor would be responsible for explicitly sending the individuals to the other processors (the “slaves”) for evaluation, collecting the results, and applying the genetic operators to produce the next generation. The number of

individuals assigned to any processor may be constant, but in some cases (like in a multi-user environment where the utilization of processors is variable) it may be necessary to balance the computational load among the processors by using a dynamic scheduling algorithm (e.g., guided self-scheduling).

Another aspect of GAs that can be parallelized is the application of the genetic operators. Crossover and mutation can be parallelized using the same idea of partitioning the population and distributing the work among multiple processors. However, these operators are so simple that it is very likely that the time required sending individuals back and forth would offset any performance gains.

The communication overhead also obstructs the parallelization of selection, mainly because several forms of selection need information about the entire population and thus require some communication. Recently, Branke parallelized different types of global selection on a 2-D grid of processors and show that their algorithms are optimal for the topology they use (the algorithms take $O(\sqrt{n})$ time steps on a $\sqrt{n} \times \sqrt{n}$ grid).

In conclusion, master-slave parallel GAs are easy to implement and it can be a very efficient method of parallelization when the evaluation needs considerable computations. Besides, the method has the advantage of not altering the search behavior of the GA, so all the theory available for simple GAs can be applied directly.

5.6.2 Island PGA

In this model, subpopulations evolve separately, and their individuals migrate among these sub-populations at every generation. The selected migrants could be either the best individual or just randomly selected ones. This model avoids the overhead of central control and presents a nature of parallelism, which makes it easy to implement this model on multiprocessor platforms and drastically speed up evolution.

In population genetics an island model is one where separate and isolated subpopulations evolve independently and in parallel. It is believed that multiple distributed subpopulations, with local rules and interactions, are a more realistic mode; of species in nature that a single large population. The island model genetic algorithm (IMGA) is analogous to the island model of population genetics. A GA population is divided into several subpopulations, each of which is randomly initialized and runs an independent sequential GA on its own subpopulation. Occasionally, fit strings migrate between subpopulations.

The migration of strings between subpopulations is a key feature of the IMGA. First, it allows the distribution and sharing of above average schemata through the strings that migrate. This serves to increase the overall selective pressure since additional reproductive trials are allocated to those strings that are fit enough to migrate. At the same time, the introduction of migrant strings into the local population helps to maintain genetic diversity, since the migrant string arrives from a different subpopulation, which has evolved independently.

The IMGA may be subject to premature convergence pressure if too many copies of a fit string migrate too often and too many subpopulations. It is possible that

after a certain number of migration steps each subpopulation contains a copy of the globally fittest individual and copies of this string migrate between subpopulations. The island model algorithm is as follows

1. *Define a suitable representation of the problem to solve, generate randomly a population of candidate solutions and partition it into Numpop subproblems; define a neighborhood structure for the subpopulations.*
2. *For each subpopulation* $Sp_i(i = 1, 2 \ldots .Numpop)$ *execute steps 3 and 4.*
3. *For Num-generations*:= 1 *to NG apply the chosen genetic operators to* Sp_i *(selection, crossover, mutation, local hill climbing, recombination etc).*
4. *Send the best individual found to the n neighbors, receive their best ones and replace thereby n individuals of the population (the worst ones, the most similar to the incoming ones, n randomly chosen etc)*
5. *If not finished, go to step2.*

The IMGA itself is a logical model. For example, an IMGA can be executed on a sequential computer by time-sharing the processor over the computations associated with each subpopulation's sequential GA. However, the most natural computer hardware on which an IMGA can be implemented is a distributed memory parallel computer. In this case each island is mapped to a node and the processor on that node runs the sequential GA on its subpopulation. Since the nodes execute in parallel, it is possible to perform more reproductive trials in a fixed (elapsed) time period as processors are added, assuming the parallel computing overheads associated with migrating strings do not increase the computational effort significantly. Because selection and other GA operators are applied locally, no global synchronization is required. Finally, strings migrate relatively infrequently, and the amount of data sent is usually small. The result is a very low communication to computation ratio.

A generic IMGA is shown in Figure 5.6. Here $P(t)$ is the population of strings at generation t. At each generation one new string is inserted into the population. The first step is to pick a random string, x_{random}, and apply a local search heuristic to it. Next two parent strings x_1 and x_2 are selected, and a random number, $r \in [0, 1]$, is generated. If r is less than the crossover probability p_c, then two new offsprings are created through crossover and one of them x_{new} is randomly selected to insert in the population. If r is not less than the crossover probability then one of the two parent strings is randomly selected, a copy of that parent is made, and mutation is applied to flip bits in the copy with probability 1/n.

In either case, the new string is tested to see whether it duplicates a string already in the population. If it does, it undergoes mutation until it is unique. A test is performed to check whether a string is to be migrated on this iteration. If so, the neighboring subpopulation to migrate the string is determined, and the string to migrate, $x_{migrate}$, is selected and sent to the neighbour. A migrant string, x_{recv}, is then received from a neighboring population, and the string to delete, x_{delete}, is determined and replaced by x_{recv}. The least-fit string in the population is deleted x_{new}, is inserted and the population is reevaluated.

5.6.3 Cellular PGA

Cellular (also known as massively parallel GAs) genetic algorithms use a spatial disposition for the individuals of one single population. Genetic operations take place in parallel for every node of the population but every individual interacts only with its neighborhood. Hence the selection, crossover and mutation take place by only considering the adjacent strings. The replacement policy destroys the considered individual by overwriting it with the newly computed string.

The algorithm for cellular model is as follows

1. *Randomly generate an initial population of candidate solutions (individuals) and assign each of them to a cell of a grid; compute the fitness of each individual.*
2. *For each current cell c of the grid execute steps 3,4,5 and 6.*
3. *Select the new individual to occupy c among those assigned to c and its neighbors.*
4. *Randomly select an individual in the neighborhood of c and recombine it with that assigned to c with probability p_c; assign one of the offspring to c.*
5. *Mutate the individual assigned to c with probability p_m.*
6. *Compute the fitness of the new individual assigned to c.*
7. *If not finished go to step 2.*

```
t←0
initialize P(t)
evaluate P(t)
for each generation
        local_search (x_random < @1;> P(t))
        select (x1, x2) from P(t)
        if (r < p_c) then
                x_new = crossover (x1, x2)
        else
                x_new = mutate (x1, x2)
        endif
        delete (x_worst ∈ P(t))
        while ()(x_new ∈ P(t))
                mutate(x_new)
        P(t+1) ← P(t) ∪ x_new
        If(migration generation) then
                t0 = neighbor(myid, gen)
                x_migrate = string_to_migrate(P(t+1))
                send_string(t0, x_migrate)
                x_recv = recv_string()
                x_delete = string_to_delete(P(t+1))
                replace_string(x_delete, x_recv, P(t+1))
        endif
        evaluate P(t+1)
        t←t+1
        end for
```

Fig. 5.6 IMGA

The optimal population size for a GA is machine dependant. Cellular GAs have been implemented on multiprocessors due to the close similarities between the model and the physical disposition of CPUs. OCCAM and different versions of C have been used in the implementation. At present, multiprocessors machines like transputers are being shifted to applications in which a sequential algorithm is executed with high speed. However cellular GA on multiprocessors is not very frequent because the parallel market has plenty of local area network (cheaper, very fast, resources sharing, etc). One intermediate approach could be the simulation of a fine-grained GA in a network of workstations. A typical cellular GA is shown in Figure 5.7.

In cellular GAs, the isolation by distance property allows a high diversity and the selection pressure is also weaker due to the local selection operator. The appearance of new species of solution in the grid (for example) and the refinement of existing solutions are both allowed and desired. The exploitation/exploration balance is then in many applications quite well developed by a parallel cellular GA. *Cellular GAs (or mpGAs) have many points in common with distributed GAs (dGAs)*

- They both use some kind of spatial topology: dGAs usually a ring and mpGAs usually locate individuals in a grid.
- They both make parallel exchanges of information among neighbors (neighbors are subpopulations of N individuals in dGAs or else just single individuals in mpGAs).
- The individuals of a traditional population can be viewed as a full-connected cellular topology.

In the above example shown in Figure 5.8 the low monitors exchange individuals in a ring topology (much like a migration dGA scheme). Each L-monitor controls a population with a given cellular (mpGA) disposition of individuals. With these considerations a unified parallel model can be implemented in the same software system and that additionally opens new questions relating the merging and cooperation among different submodels.

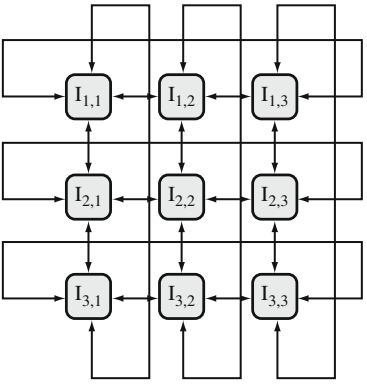


Fig. 5.7 Cellular Model

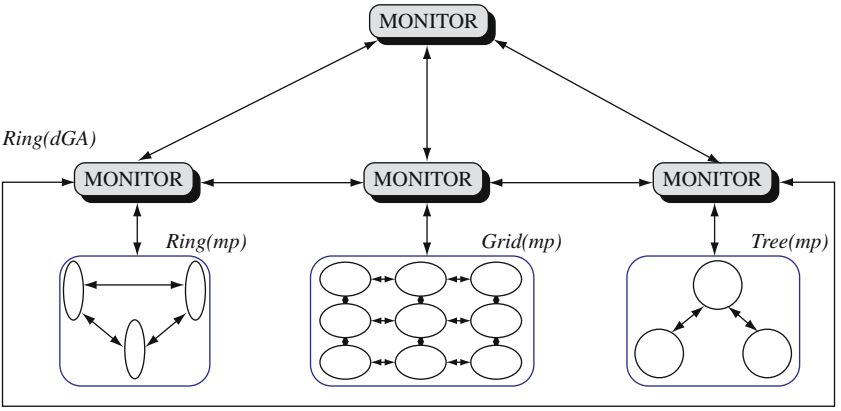


Fig. 5.8 The dGAME model. Merging of models can yield more robust searches. There exist two levels of search: a dGA of mpGAs

5.7 PGA Models Based on Parallelism

The PGA models based on the granularity of parallelism are discussed below.

5.7.1 Global with Migration (coarse-grained)

Typical coarse grain models consists a set of parallel subpopulations that evolve independently from each other and that periodically exchange individuals among them. The topology for the exchanges is usually a ring or else some similar disposition. The subpopulations undergo exactly the same genetic operations on the same kind of genotype and the result is the best individual found in the overall population.

In a coarse grained parallel genetic algorithm (CPGA), multiple processors each run a sequential GA on their own subpopulation. Processors exchange strings from their subpopulation with other processors. Some important choices in a CPGA are which other processors a processor exchanges strings with, how often processors exchange strings, how many strings processors exchange with each other and what strategy is used when selecting strings to exchange.

5.7.2 Global with Migration (fine-grained)

The development of massively parallel computers triggers a new approach of parallel genetic algorithms. To take advantage of new architectures with even a greater number of processors and les communication costs, fine-grained PGAs have been developed. The population is now partitioned into a large number of very small

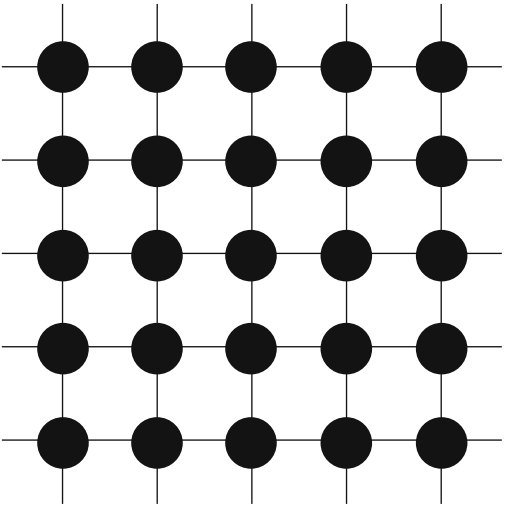


Fig. 5.9 Schematic of A Fine-Grained Parallel GA

subpopulations as shown in Figure 5.9. The limit (and may be ideal) case is to have just one individual for every processing element available. Basically, the population is mapped onto a connected processor graph, usually, one individual on each processor. Mating is only possible between neighboring individual, i.e, individuals stored on neighboring processors. The selection is also done in a neighborhood of each individual and so depends only on local information. A motivation behind local selection is biological. In nature there is no global selection, instead natural selection is a local phenomenon, taking place in an individual's local environment. In order to compare this model to the island model, each neighborhood can be considered as a different deme. But here, the demes overlap providing a way to disseminate good solutions across the entire population. Thus, the topology does not need to explicitly define migration roads and migration rate.

It is common to place the population on a 2-D or 3-D grid because in many massively parallel computers the processing elements are connected using this topology. Consequently each individual has four neighbors. Experimentally, it seems that good results can be obtained using a topology with a medium diameter and neighborhoods not too large. Like the coarse grained models, it worth trying to simulate this model even on a single processor to improve the results. Indeed, when the population is stored in a grid like this, after few generations, different optima could appear in different places on the grid.

Fine-grained parallel GA has only one population, but it has a spatial structure that limits the interactions between individuals. An individual can only compete and mate with its neighbors, but since the neighborhoods overlap good solutions may disseminate across the entire population.

Robertson parallelized the genetic algorithm of a classifier system on a Connection Machine. He parallelized the selection of parents, the selection of classifiers to replace, mating, and crossover. The execution time of his implementation was independent of the number of classifiers (up to 16K, the number of processing elements in the CM-1).

5.8 Communication Topologies

A traditionally neglected component of parallel GAs is the topology of the interconnection between demes. The topology is an important factor in the performance of the parallel GA because it determines how fast (or how slow) a good solution disseminates to other demes. If the topology has a dense connectivity (or a short diameter, or both) good solutions will spread fast to all the demes and may quickly take over the population. On the other hand, if the topology is sparsely connected (or has a long diameter), solutions will spread slower and the demes will be more isolated from each other, permitting the appearance of different solutions. These solutions may come together at a later time and recombine to form potentially better individuals.

The communication topology is also important because it is a major factor in the cost of migration. For instance, a densely connected topology may promote a better mixing of individuals, but it also entails higher communication costs.

The general trend on multi-deme parallel GAs is to use static topologies that are specified at the beginning of the run and remain unchanged. Most implementations of parallel GAs with static topologies use the native topology of the computer available to the researchers. For example, implementations on hypercubes and rings are common.

A more recent empirical study showed that parallel GAs with dense topologies find the global solution using fewer function evaluations than GAs with sparsely connected ones. This study considered fully connected topologies, 4-D hypercubes, a 4×4 toroidal mesh, and unidirectional and bidirectional rings.

The other major choice is to use a dynamic topology. In this method, a deme is not restricted to communicate with some fixed set of demes, but instead the migrants are sent to demes that meet some criteria. The motivation behind dynamic topologies is to identify the demes where migrants are likely to produce some effect. Typically, the criteria used to choose a deme as a destination includes measures of the diversity of the population or a measure of the genotypic distance between the two populations (or some representative individual of a population, like the best).

5.9 Hierarchical Parallel Algorithms

A few researchers have tried to combine two of the methods to parallelize GAs, producing hierarchical parallel GAs. Some of these new hybrid algorithms add a new degree of complexity to the already complicated scene of parallel GAs, but other hybrids manage to keep the same complexity as one of their components. When two methods of parallelizing GAs are combined they form a hierarchy. At the upper level most of the hybrid parallel GAs are multiple-population algorithms. Some hybrids have a fine-grained GA at the lower level as shown in Figure 5.10. For example Gruau invented a "mixed" parallel GA. In his algorithm, the population of each deme was placed on a 2-D grid, and the demes themselves were connected as a 2-D torus. Migration between demes occurred at regular intervals, and good results were reported for a novel neural network design and training application.

At the lower level the migration rate is faster and the communication topology is much denser than at the upper level.

ASPARAGOS was updated recently, and its ladder structure was replaced by a ring, because the ring has a longer diameter and allows a better differentiation of the individuals. The new version (Asparagos96) maintains several subpopulations that are themselves structured as rings. Migration across subpopulations is possible, and when a deme converges it receives the best individual from another deme. After all the populations converge, Asparagos96 takes the best and second best individuals of each population, and uses them as the initial population for a final run.

Lin, Goodman, and Punch also used a multi-population GA with spatially-structured subpopulations. Interestingly, they also used a ring topology — which

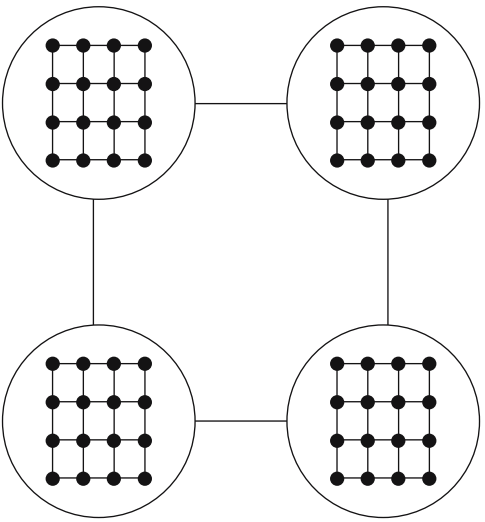


Fig. 5.10 Hierarchical GA with a multi-deme GA at the upper level and a fine-grained GA at the lower level

is very sparse and has a long diameter — to connect the subpopulations, but each deme was structured as a torus. The authors compared their hybrid against a simple GA, a fine-grained GA, two multiple-population GAs with a ring topology (one uses 5 demes and variable deme sizes, the other uses a constant deme size of 50 individuals and different number of demes), and another multi-deme GA with demes connected on a torus. Using a job shop scheduling problem as a benchmark, they noticed that the simple GA did not find solutions as good as the other methods, and that adding more demes to the multiple-population GAs improved the performance more than increasing the total population size. Their hybrid found better solutions overall.

Another type of hierarchical parallel GA uses a master-slave on each of the demes of a multi-population GA as seen in Figure 5.11. Migration occurs between demes, and the evaluation of the individuals is handled in parallel. This approach does not introduce new analytic problems, and it can be useful when working with complex applications with objective functions that need a considerable amount of computation time. Bianchini and Brown presented an example of this method of hybridizing parallel GAs, and showed that it can find a solution of the same quality of a master-slave parallel GA or a multi-deme GA in less time.

Interestingly, a very similar concept was invented by Goldberg in the context of an object-oriented implementation of a "community model" parallel GA. In each "community" there are multiple houses where parents reproduce and the offspring are evaluated. Also, there are multiple communities and it is possible that individuals migrate to other places.

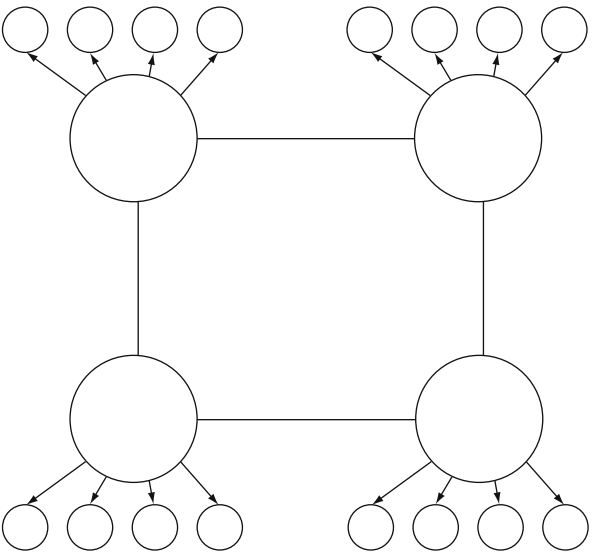


Fig. 5.11 A hierarchical parallel GA with multi-deme GA at the upper level where each node is a master-slave GA

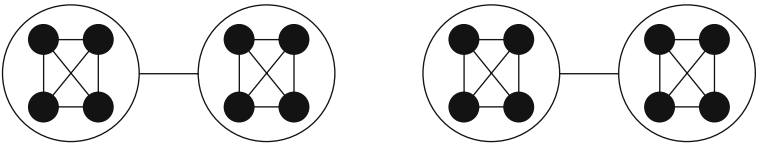


Fig. 5.12 Hybrid PGA with multi-deme GAs at both the upper and the lower levels

A third method of hybridizing parallel GAs is to use multi-deme GAs at both the upper and the lower levels as shown in Figure 5.12. The idea is to force panmictic mixing at the lower level by using a high migration rate and a dense topology, while a low migration rate is used at the high level. The complexity of this hybrid would be equivalent to a multiple-population GA if the groups of panmictic subpopulations are considered as a single deme. This method has not been implemented yet.

Hierarchical implementations can reduce the execution time more than any of their components alone. For example, consider that in a particular domain the optimal speed-up of a master-slave GA is Sp_{ms} and the optimal speed-up of a multiple-deme GA is Sp_{md}. The overall speed-up of a hierarchical GA that combines these two methods would be $Sp_{ms} \times Sp_{md}$.

5.10 Object Orientation (OO) and Parallelization

Traditional imperative languages like C or PASCAL endow the implementation with a high computational speed. However, more actual technologies such as object oriented design can have an outstanding impact on the quality of the implementation. Among the benefits of using the OO methodology for a PGA implementation, the following ones can be mentioned:

- Reusing the software is at present difficult: a software tool is either too specific or thus useless for other problems or it is too general and thus inefficient. OO can help in reusing software, in the fast building of prototypes and also in the security of this software.
- A common architecture of classes for a genetic system could be created and enriched by different people and experience. Comparisons should then be readily simple and meaningful.
- An object oriented implementation of a PGA can also help experimenting and combining different granularities, topologies, operators, hybridization, competition or cooperation with other techniques and offers many other added facilities to create new models of evolution.

5.11 Recent Advancements

This section summarizes some recent advancement in the theoretical study of parallel GAs. First, the section presents a result on master-slave GAs, and then there is a short presentation of a deme sizing theory for multiple-population GAs.

An important observation on master-slave GAs is that as more processors are used, the time to evaluate the fitness of the population decreases. But at the same time, the cost of sending the individuals to the slaves increases. This tradeoff between diminishing computation times and increasing communication times entails that there is an optimal number of slaves that minimizes the total execution time. A recent study concluded that the optimal is

$$s^* = \sqrt{\frac{nT_f}{T_c}} \tag{5.2}$$

where s^* is the population size, T_f is the time it takes to do a single function evaluation, and T_c is the communications time. The optimal speed-up is 0.5 s^*.

A natural starting point in the investigation of multiple-population GAs is the size of the demes, because the size of the population is probably the parameter that affects most the solution quality of the GA. Recently, Cantú-Paz and Goldberg extended a simple GA population sizing model to account for two bounding cases of coarse-grained GAs. The two bounding cases were a set of isolated demes and a set of fully connected demes. In the case of the connected demes, the migration rate is set to the highest possible value.

The population size is also the major factor to determine the time that the GA needs to find the solution. Therefore, the deme sizing models may be used to predict the execution time of the parallel GA, and to compare it with the time needed by a serial GA to reach a solution of the same quality. Cantú-Paz and Goldberg integrated the deme sizing models with a model for the communications time, and predicted the expected parallel speed-ups for the two bounding cases.

There are three main conclusions from this theoretical analysis. First, the expected speed-up when the demes are isolated is not very significant as seen in Figures 5.13 and 5.14. Second, the speed-up is much better when the demes communicate. And finally, there is an optimal number of demes (and an associated deme size) that maximizes the speed-up as shown in Fig 5.15.

The idealized bounding models can be extended in several directions, for example to consider smaller migration rates or more sparsely connected topologies. The fact that there is an optimal number of demes limits the processors that can be used to reduce the execution time. Using more than the optimal number of demes is wasteful, and would result in a slower algorithm. Hierarchical parallel GAs can use more processors effectively and reduce the execution time more than a pure multiple-deme GA.

As GAs are applied to larger and more difficult search problems, it becomes necessary to design faster algorithms that retain the capability of finding acceptable

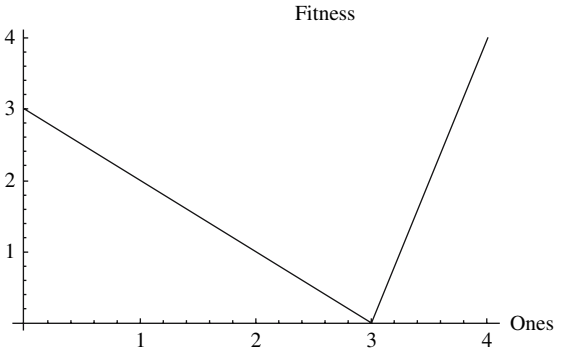


Fig. 5.13 A deceptive 4-bit trap function of unity. The horizontal axis is the number of bits set to 1 and the vertical axis is the fitness value. The difficulty of this function can be varied easily by using more bits in the basin of the deceptive optimum, or by reducing the fitness difference between the global and deceptive maxima. The first test problem was constructed by concatenating 20 copies of this function and the second test problem is formed with 20 copies of a similar deceptive function of 8 bits

solutions. Parallel GAs are capable of combining speed and efficacy, and that a better understanding is reached which should allow to utilize them better in the future.

Parallel GAs are very complex and, of course, there are many problems that are still unresolved. A few examples are: (1) to determine the migration rate that makes distributed demes behave like a single panmictic population, (2) to determine an adequate communications topology that permits the mixing of good solutions, but that does not result in excessive communication costs, (3) find if there is an optimal number of demes that maximizes reliability.

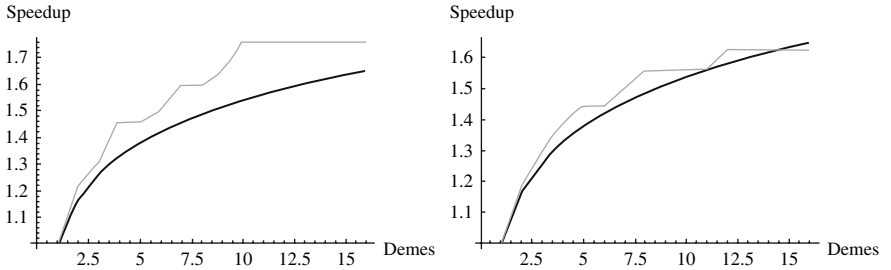
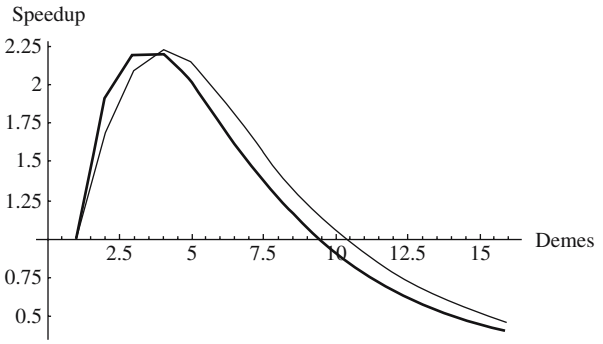
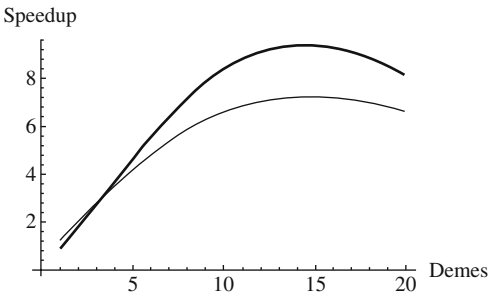


Fig. 5.14 Projected and experimental speed-ups for test functions with 20 copies of 4-bit and 8-bit trap functions using from 1 to 16 isolated demes. The thin lines show the experimental results and the thick lines are the theoretical predictions. The quality demanded was to find at least 16 copies correct



(a) 4-bit trap



(b) 8-bit trap

Fig. 5.15 Projected and experimental speed-ups for test functions with 20 copies of 4-bit and 8-bit trap functions using from 1 to 16 fully connected demes. The quality requirement was to find at least 16 correct copies

5.12 Advantages of Parallel Genetic Algorithms

A PGA has the same advantages as a serial GA, consisting in using representations of the problem parameters (and not the parameters themselves), robustness, easy customization for a new problem, and multiple-solution capabilities. These are the characteristics that led GAs and other EAs to be worth of study and use. In addition, a PGA is usually faster, less prone to finding only sub-optimal solutions, and caable of cooperating with other search techniques in parallel.

- Works on a coding of the problem (least restrictive)
- Easy parallelization as islands or neighborhoods
- Basically independent of the problem (robustness)
- Better search, even if no parallel hardware is used
- Can yield alternative solutions to the problem
- Higher efficacy than sequential GAs

- Parallel search from multiple points in the space
- Easy cooperation with other search procedures

Summary

- Parallel genetic algorithms can be classified into global master-slave parallelization, fine-grained algorithms, multiple-deme, and hierarchical parallel GAs.
- Parallel population model describes basically, how the population is divided into subpopulations and how the information exchange between these subpopulations is organized.
- The parallel population models are classified into global, regional and local models based on the radius of their selection operator.
- Depending on the distribution of the population, PGAs are classified into Centralized and Distributed PGA
- PGAs are classified into Master – Slave PGA, Island PGA and Cellular PGA based on the Implementation.
- Coarse grain and fine grain models are based on the degree of granularity.
- Two of the methods can be combined to parallelize GAs, producing hierarchical parallel GAs.
- PGA is much efficient than sequential GAs and can yield alternative solutions to the problem

Review Questions

1. What is speedup?
2. Explain the need for Parallel GA.
3. What are the types of PGA based on the Parallelism?
4. What are hierarchical parallel genetic algorithms?
5. Which structures play a vital role in classification of Parallel genetic algorithms?
6. What is metacomputing?
7. Is there any other advantage of employing PGA other than speedup?
8. Discuss the PGA models based on distribution of population.
9. What is the advantage of parallelizing the evaluation of individuals?
10. How can parallelism be applied for genetic operators?
11. What is cellular PGA?
12. Define granularity.
13. Differentiate regional and local models.
14. What is migration topology?
15. Explain the classification of global population models.
16. What are the parameters that control the exchange of individuals?

17. Explain the migration strategies.
18. Write an algorithm for cellular PGA.
19. What is the role of communication topology in PGA?
20. Explain the strategies and parameters for local population models.

Chapter 6
Applications of Evolutionary Algorithms

Learning Objectives: On completion of this chapter the reader will have knowledge on applications of Evolutionary algorithms such as:

- Fingerprint Recognizer
- Automatic Engineering Design
- Tool for Grammar Development
- Waveform Generation
- Scheduling Earth Observing Satellites
- Scenario-Based Risk-Return Portfolio Optimization For General Risk Measures

6.1 A Fingerprint Recognizer using Fuzzy Evolutionary Programming

6.1.1 Introduction

Fingerprints have been used for many centuries as a means of identifying people. As it is well known that fingerprints of an individual are unique and are normally unchanged during the whole life, the use of fingerprints is considered one of the most reliable methods of personal verification. This method has been widely used in criminal identification, access authority verification, financial transferring confirmation, and many other civilian applications.

In the olden days, fingerprint recognition was done manually by professional experts. But this task has become more difficult and time consuming, particularly in the case where a very large number of fingerprints are involved. During the past decade, several automatic fingerprint identification systems have been made available to meet the demand of new applications. The methods used in these systems are still far from complete satisfaction, however, due to inaccurate extraction of fingerprint characteristics and ineffective pattern matching procedures, which are the two major tasks of fingerprint identification. The Federal Bureau of Investigation's

method of identifying a fingerprint by its set of minutiae is widely used in automatic fingerprint identification systems. However, the way of extracting a fingerprint's minutiae differs from system to system. Many systems require some form of image preprocessing such as transforming the fingerprint into a binary image and trimming the image's ridges into single pixel lines, before the detection of minutiae is carried out. This may cause some loss of information and result in inaccurate detection of the fingerprint's minutiae. The task of matching an input fingerprint's minutiae with those from a database is much more difficult. The existing methods that are based on mathematical approximation and string matching algorithms or on relaxation and simulated annealing have shown to be rather ineffective and time consuming. The method of discrete Hough transform is obviously inappropriate as the search space is continuous. These difficulties stem from possible skin elasticity, different scales, and difference in positions of fingerprints.

In this application, the method of direct gray scale minutiae detection is improved by a backup validating procedure to eliminate false minutiae. Fuzzy evolutionary programming technique which has been used successfully in speaker identification, image clustering is employed for minutiae matching. Experimental results show this approach is also highly effective in fingerprint recognition.

6.1.2 Fingerprint Characteristics

A fingerprint is a textural image containing a large number of ridges that form groups of almost parallel curves (Figure 6.1). It has been established that fingerprint's ridges are individually unique and are unlikely to change during the whole life.

Although the structure of ridges in a fingerprint is fairly complex, it is well known that a fingerprint can be identified by its special features such as *ridge endings, ridge bifurcation, short ridges*, and *ridge enclosures*. These ridge features are collectively

Fig. 6.1 A Fingerprint

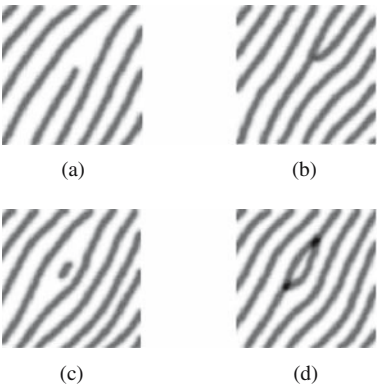


Fig. 6.2 (a) Ridge ending; (b) Ridge bifurcation; (c) short ridge; (d) Ridge enclosure

called the *minutiae* of the fingerprint. For automatic detection of a fingerprint, it suffices to focus on two types of minutiae, namely *ridge endings* and *bifurcation*. Figure 6.2 shows the forms of various minutiae of a fingerprint.

A full fingerprint normally contains 50 to 80 minutiae. A partial fingerprint may contain fewer than 20 minutiae. According to the Federal Bureau of Investigation, it suffices to identify a fingerprint by matching 12 minutiae, but it has been reported that in most cases 8 matched minutiae are enough.

Minutiae Extraction

For convenience, a fingerprint image is represented in reverse gray scale. That is, the dark pixels of the ridges are assigned high values where as the light pixels of the valleys are given low values. Figure 6.3 shows a section of ridges in this representation.

In a fingerprint, each minutia is represented by its location (x, y) and the local ridge direction φ. Figure 6.4 shows the attributes of a fingerprint's minutia. The process of minutiae detection starts with finding a summit point on a ridge, then

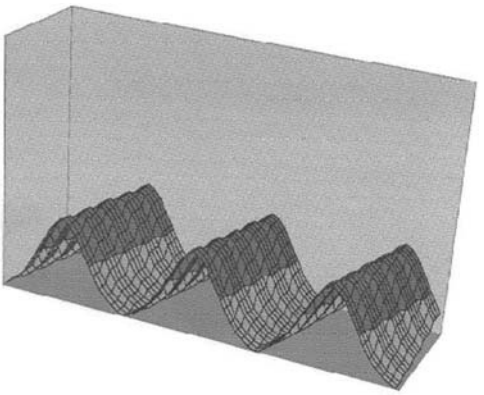


Fig. 6.3 Section of ridges

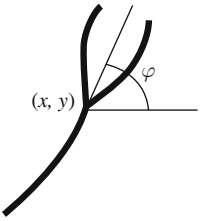


Fig. 6.4 A minutia's attributes

continues by tracing the ridge until a minutia, which can be either a ridge ending or bifurcation, is encountered. Details of these tasks are described in the following subsections.

Finding a Ridge Summit Point

To find a summit point on a ridge, start from a point $x = (x1,\ x2)$ and compute the direction angle φ by using the gradient method (Figure 6.5). Then the vertical section orthogonal to the direction φ is constructed (To suppress light noise, the section gray values are convoluted with a Gaussian weight function). The point in this section with maximum gray level is a summit point on the nearest ridge.

The direction angle φ at a point x mentioned above is computed as follows. A 9×9 neighborhood around x is used to determine the trend of gray level change. At each pixel $u = (u1,\ u2)$ in this neighborhood, a gradient vector $v(u) = (v1(u),\ v2(u))$ is obtained by applying the operator $h = (h1,\ h2)$ with

$$h_1 = \frac{1}{4}\begin{bmatrix} -1 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix} \qquad h_2 = \frac{1}{4}\begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & -1 \end{bmatrix}$$

to the gray levels in a neighborhood of u. That is,

$$v_1(u) = \sum g(y)h_1(y-u), v_2(u) = \sum g(y)h_2(y-u) \tag{6.1}$$

where y runs over the eight neighboring pixels around u and $g(y)$ is the gray level of pixel y in the image. The angle φ represents the direction of the unit vector t that is (almost) orthogonal to all gradient vectors v. That is, t is chosen so that $\Sigma(v.t)^2$ is minimum.

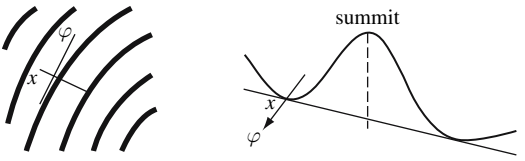


Fig. 6.5 A summit point on a ridge

Tracing a Ridge

The task of tracing a ridge line (Figure 6.6) to detect minutiae is described in the following algorithm. This algorithm also constructs a traced image of the fingerprint. Every time a new summit point of the ridge is found, its location in the traced image is assigned a high gray value and the surrounding pixels are given lower gray levels if they have not been marked.

Algorithm 1 (Ridge Tracing)

Start from a summit point x of a ridge.

Repeat

Compute the direction angle φ at x; Move μ pixels from x along the direction φ to another point y; Find the next summit point z on the ridge, which is the local maximum of the section orthogonal to direction φ at point y; Set $x = z$;

Until point x is a termination point (i.e. a minutia or off valid area).
Determine if the termination point x is a valid minutia, if so record it.

End Algorithm 1

There are three criteria used to terminate tracing a ridge. The first stopping condition is that when the current point is out of the area of interest. That is, the current point is within 10 pixels from the border, as experiments show that there are rarely any minutiae close to the edges of the image. The second criterion determines a ridge ending: the section at the current point contains no pixels with gray levels above a pre-specified threshold. In this case, the previous point on the ridge is recorded as a ridge endpoint. The last stopping condition corresponds to the case of a possible bifurcation: the current point is detected to be on another ridge that has been marked on the traced image.

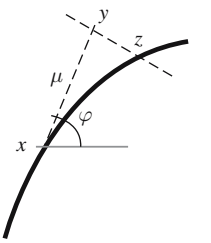


Fig. 6.6 Ridge tracing

Algorithm 1 is backed up by a checking procedure that determines if a termination point is a valid minutia. The procedure is expressed as follows.

Algorithm 2 (Elimination of False Minutiae)

If the current ridge end is close to another ridge end with almost opposite direction, then delete both of them and join the gap, as they are simply broken ends of the same ridge.
If the current bifurcation point is close to the end of one of its branch, then delete both of them, as short branch of a bifurcation is normally a result of light noise in the image.
If the current termination is close to more than two other terminations, then delete all of them, as they are likely caused by damaged ridges in the image.

The above algorithms form one major component of the fingerprint recognizing system, called the *Fingerprint Administrator*. Figure 6.7 depicts the user-interface feature of the *Fingerprint Administrator*. The input fingerprint image is displayed in the left box, and the result of ridge tracing and detection of minutia is shown in a traced image in the right box. Observe that in the traced image, the ridge summits are shown in black color, their surrounding of five pixels is colored red, and the detected minutiae are marked with yellow tangent vectors. If the *Save* button is clicked, the coordinates of the detected minutiae and their associated direction angles are saved in a database in the form of linked list.

The *Fingerprint Administrator* is used to extract minutiae of known fingerprints and store them in a database. It is also used to extract minutiae of an input fingerprint for the purpose of identification.

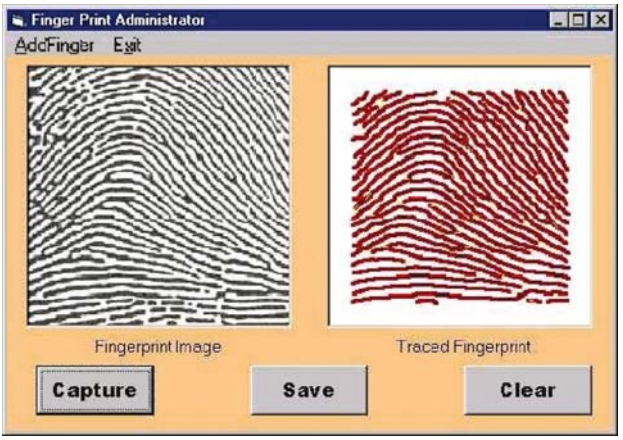


Fig. 6.7 Fingerprint administrator's user-interface

6.1.3 Fingerprint Recognition using EA

The primary purpose of the fingerprint recognizing system is to calculate the matching degree of the target fingerprint with the images in a database and to decide if it belongs to a particular individual. A fingerprint is said to match one image in the database if the degree of matching between its minutiae and that of the image in the database is higher than some pre-specified acceptance level. The method of calculating this matching degree is based on fuzzy evolutionary programming technique, which is described below.

Consider two fingerprints that are represented by their sets of minutiae $P = \{p_1, \mathrm{K}, p_m\},\ Q = \{q_1, \mathrm{K}, q_n\}$, where $p_i = (x_i,\ y_i, \alpha_i)$ and $q_j = (u_j,\ v_j,\ \beta_j)$, for $1 \le i \le m,\ 1 \le j \le n$. Observe that the two sets may not have the same number of points, and that the order of the points in each set is possibly arbitrary.

The principal task is to find a transformation $F = (s,\ \theta,\ \delta x,\ \delta y)$ that transforms the set of minutiae P into the set Q.

Here, s represents a scaling factor, θ an angle of rotation, and $(\delta x,\ \delta y)$ a translation in the xy-plane. Thus, the transform $F(p) = (x', y', \alpha')$ of a minutia $p = (x,\ y,\ \alpha)$ is defined by (Figure 6.8):

$$\begin{bmatrix} x' \\ y' \\ \alpha' \end{bmatrix} = s \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & \frac{1}{s} \end{bmatrix} \begin{bmatrix} x \\ y \\ \alpha \end{bmatrix} + \begin{bmatrix} \delta x \\ \delta y \\ \theta \end{bmatrix}$$

Also, in order to calculate the degree of matching, $F(p)$ a fuzzy set (also denoted by $F(p)$ for convenience) the membership function of which is defined by:

$$\mu_{F(p)}(q) = \min(\mu_{(0,\varepsilon)}(d(F(p), q)),\ \mu_{(0,\varepsilon')}(|\ \alpha - \beta + \theta|)), \tag{6.2}$$

is associated with each minutia, where,

$\mathtt{d(F(p), q)} = \sqrt{(x-u)^2 + (y-v)^2}$ and $\mu_{(0,\varepsilon)}$ represents a fuzzy subset of the real line defined as follows:

$$\mu_{(0,\varepsilon)}(x) = \begin{cases} \frac{e^{-p\left(\frac{x}{\varepsilon}\right)^2} - e^{-p}}{1 - e^{-p}} & \text{if } |x| \le \varepsilon \\ 0 & \text{otherwise} \end{cases} \tag{6.3}$$

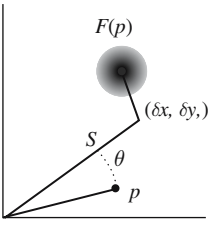


Fig. 6.8 $F(p)$ is transform of p

Intuitively some degree of tolerance is allowed in matching the minutiae $F(p)$ and q, but this tolerance decreases rapidly when the two minutiae are far apart. The matching degree between two sets $F(P)$ and Q is defined as:

$$\sum_{i=1}^{m}\sum_{j=1}^{n}\mu_{F(p_i)}(q_j) \tag{6.4}$$

The task of finding a transformation F to match two sets of minutiae P and Q consists of two phases. First the rotation angle θ is estimated by the following algorithm.

Algorithm 3 (Estimation of Rotation)

Divide the interval $[-\pi, \pi]$ into K subintervals $[\theta_k, \theta_{k+1}]$, $k = 1, \ldots, K$.

Set up an integer array $c[1, K]$ and a real array $\Theta[1, K]$ and initialize them to 0.

For each $i = 1, \ldots, m$ and each $j = 1, \ldots, n$ do

Find an index k such that $\theta k \leq \beta j - \alpha i < \theta k + 1$;

Increment $c[k]$ by 1 and increment $\Theta[k]$ by $(\beta j - \alpha i)$.

Find the index k^* such that $c[k^*]$ is maximum.

Let θ be defined by

$$\theta = \frac{\Theta[k^*]}{c[k^*]}$$

Having established the rotation angle θ, the remaining parameters s and δ_x, δ_y of the transformation F are estimated by the following algorithm.

Algorithm 4 (Fuzzy Evolutionary Programming)

Generate a population of m chromosomes

$F_k = (s_k, \theta, \delta x_k, \delta y_k)$, $k = 1, \ldots, m$, where the parameter values are randomly taken from appropriate intervals.

For each $k = 1, \ldots, m$, compute the fitness:

$$f_k = \sum_{i=1}^{m}\sum_{i=1}^{n}\mu_{F_k(p_i)}(q_j)$$

For each $k = 1, \ldots, m$, generate an offspring $F_{m+k} = (s_{m+k}, \theta, \delta x_{m+k}, \delta y_{m+k})$ as follows:

$$s_{m+k} = s_k + \rho_1(s_k)N(0, \sigma(f_k)),$$

$$\delta x_{m+k} = \delta x_k + \rho_2(\delta x_k)N(0,\sigma(f_k)),$$
$$\delta y_{m+k} = \delta y_k + \rho_3(\delta y_k)N(0,\sigma(f_k)),$$

where $N(0,\sigma(f_k))$ is a normal distribution with variance $\sigma(f_k)$ inversely proportional to the fitness f_k, and the factors $\rho_1(s_k)$, $\rho_2(\delta x_k)$ and $\rho_3(\delta y_k)$ are used to ensure that the new values are within the predetermined intervals.

Compute the fitness f_{m+k} of the new offspring using formula (). For each $k = 1,\ldots,\ 2m$, select a random set U of c indices from 1 to $2m$, and record the number wk of $h\in U$ such that $f_h \leq f_k$.

Select m chromosomes from the set $\{F_1,\ K,\ F_{2m}\}$ that have highest scores wk to form the next generation of population.

Until the population is stabilized.

Algorithms 3 and 4 form the principal component of the system: the *fingerprint recognizer*. The fingerprint recognizer receives an input fingerprint, calls the *fingerprint administrator* to extract the fingerprint's minutiae, then tries to match this set of minutiae with those in the database, until either a good match is found, or the database is exhausted.

6.1.4 Experimental Results

A number of fingerprints of various types, including plain and tented arch, ulna and radial loop, plain whorl, central pocket whorl, double whorl, and accidental whorl, are fed to the system's *fingerprint administrator* for minutiae detection. The resulting traced images (as depicted in Figure 6.7) are manually inspected and the results are shown in Table 6.1.

Note that the fingerprints' ridges are accurately traced, broken ridges are effectively rejoined, and the short ridges and spur ridges are correctly adjusted. Manual checking confirms that most of those anomalies in the fingerprints are result of light noise, skin cut or distortion. The adjustment was realized by the system's process of eliminating false minutiae (Algorithm 2).

A number of fingerprints are taken from the database and are modified by various rotations, resizing, and shifting, for use in testing experiments. Random light noise is added to the test images. The results of the *fingerprint recognizer's* performance are summarized in Table 6.2.

Table 6.1 Results of minutiae detection

Image type	Correct %	False %	Missed %
Arch	96	3	1
Loop	93	4	3
Whorl	92	6	2

Table 6.2 Results of minutiae matching

Test cases	Matching result %	Running time (sec)
Two identical sets of minutiae	100	12
One is a re-scale of the other	98.4	16
One is a rotation of the other	97.6	17
One is a translation of other	99.5	17
One is a transform of the other	96.2	18

6.1.5 Conclusion and Future Work

A fingerprint recognizing system that uses the method of gray scale ridge tracing backed up by a validating procedure to detect fingerprint's minutiae and that employs the technique of fuzzy evolutionary programming to match two sets of minutiae in order to identify a fingerprint is presented. The experimental results show that the system is highly effective with relatively clean fingerprints. However, for poorly inked and badly damaged fingerprints, the system appears to be not so successful. In order to handle those bad types of fingerprints, a preprocessing component that adopts the fuzzy evolutionary approach can be added to reconstruct and enhance the fingerprints before they are processed by the system. Also, it is possible to connect the system with a live fingerprint scanner that obtains a person's fingerprint directly and sends it to the system for identification.

6.2 An Evolutionary Programming Algorithm for Automatic Engineering Design

6.2.1 Introduction

In the engineering design process there is a demand for the capability to perform automatic optimization, minimizing or maximizing some derived quantity, a measure of "fitness" of the design. For real-world problems these objective function values are often computed using sophisticated and realistic numerical simulations of physical phenomena. This is an extremely computationally intensive process when models must be run tens, or maybe hundreds, of times to effectively search design parameter space. Current High Performance Computing systems mostly derive their capacity from parallel architectures so for optimization methods to be practical and effective the algorithms used must preferably have a large degree of concurrency. Figure 6.9 offers a simple taxonomy of methods that can be used for this automatic optimization.

Gradient descent algorithms, based on classical, Newtonian methods of analysis, generally exhibit rapid local convergence but due to the iterative steps of gradient determination and some form of line search most employ, they are inherently

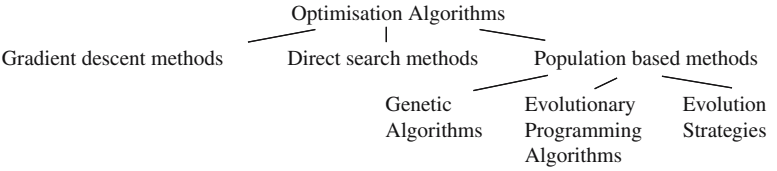


Fig. 6.9 Taxonomy of optimization algorithms

sequential in nature. In many problems arising in real-world design processes there are limits imposed on problem parameters by their nature; it does not make sense, for example, for a physical component to have negative size. The authors have previously exploited this simple bounding of engineering problems to enhance the parallelism of line searching to some extent.

Of greater potential concurrency are direct search methods, such as the Nelder-Mead Simplex algorithm. These methods do not explicitly use approximate gradient information, but instead take objective function values from a set of sample points, and use that information to continue sampling. Use of the Nelder-Mead simplex algorithm remains current, largely because on a range of practical engineering problems it is capable of returning a very good result. The early direct search methods, however, exhibit very limited concurrency. It is only over the past decade that the computational capacity available to scientists and engineers has increased to the point where population-based methods of optimization have become practical for the solution of real-world problems. Moreover, in engineering design, the evaluation of the objective function is so much slower than the rest of the algorithm, that such codes demonstrate excellent speedup in spite of the need for global communication on each iteration. Genetic Algorithms (GA) now may be frequently encountered in application to engineering problems. These have inherent, easily exploitable parallelism, and attractive global convergence probabilities, but have been considered generally slow to converge.

In the general case, evolutionary computation seeks to use insight into natural processes to inform population-based computational methods. The theory of self-organized criticality gives an insight into emergent complexity in nature. Bak contended that the critical state was the most efficient state that can actually be reached dynamically". In this state, a population in an apparent equilibrium evolves episodically in spurts, a phenomenon known as punctuated equilibrium. Local change may affect any other element in the system, and this delicate balance arises without any external, organizing force.

These concepts have been utilized in developing a new Evolutionary Programming algorithm that, in addition to the conventional mutation and selection operations, implements a further selection operator to encourage the development of a self-organized critical system. The algorithm is evaluated on a range of test cases drawn from real-world problems, and compared against a selection of algorithms, including gradient descent, direct search methods and a genetic algorithm.

6.2.2 EPSOC: An Evolutionary Programming Algorithm using Self-Organized Criticality

The general optimization problem for an arbitrary non-linear function f can be stated as:

$$\text{Minimize } f(x);\ f\ :< \text{n!} < 1 \text{ and } \text{x} = \text{fx}_0;\ :::;\ \text{x}_\text{i};\ :::;\ \text{x}_\text{ng};\ \text{x}_\text{i}\ 2 <: \tag{6.5}$$

For a population-based method, the population, *p*, consists of a set of parameter vectors, x : p = fx0; :::; xi; :::; xmg. For real-world engineering design problems, values for f(x) generally can only be derived from execution of complex numerical simulations requiring considerable computation time. Acceptable parameter values are bounded in the majority of cases, due to physical constraints of the problem considered. EPSOC treats all parameters as simply bounded.

As stated in the introduction, EPSOC is built on the basis of an Evolutionary Programming algorithm, with an additional selection operator following the method of Bak. The steps of the algorithm are outlined below:

1. Initialise a random, uniformly-distributed population, p, and evaluate each trial solution, $\text{x}_\text{i}\ \forall$ i.
2. Sort the population by objective function value, f(x).
3. Select a set, B, of the nbad worst members of the population. For each member of B, add to the set its two nearest neighbours in parameter space that are not already members of the set, or in the best half of the sorted population.
4. Apply a random, uniformly-distributed mutation to the selected set, B, i.e. re-initialise them. For all other members of the population, generate a child" by applying a small (10% of parameter range), random, uniformly distributed mutation to the "parent" member.
5. Evaluate each new trial solution, f(x).
6. If a child has a better objective function value than its parent, replace the parent with the child.
7. Repeat from step 2 until a preset number of iterations have been completed.

As each set of parameters defining a trial solution is independent of all others, it is immediately apparent that the evaluation of trial solutions at steps 1 and 5 can be performed concurrently. Since the evaluation of the objective function completely dominates the execution time, high efficiency can be expected from parallelism as per Amdahl's Law.

Critical states of evolutionary models exhibit a power-law distribution of the size and frequency of extinction events. This is apparent when the number of events involving given number of members of the population approximate a straight line in a double-log statistical plots of the phenomena. In earlier applications of self-organized criticality to optimization, it has been proposed that a separately computed power-law extinction rate be imposed on a spatial diffusion model, or cellular GA. In Krink and Thomsen's model, population members are chosen at random for

extinction, and the algorithm is greedy to the extent of maintaining the single best member, elite of one. Their algorithm apparently does not attempt to evolve a population in a critical state, but indirectly imposes the observed behaviour of such a population.

In contrast, EPSOC is a straightforward implementation of Bak's nearest neighbour, punctuated equilibrium model as an optimization algorithm. By considering the trial solution parameter vectors as defining a location in an n- dimensional parameter space, the spatial behaviour of Bak's model is realized naturally. A high degree of greediness is applied to the algorithm. Maintaining a large "elite" (in EPSOC, half the total population) can be viewed as a "constructive" operator. Like "Maxwell's demon", it accretes information and encourages a gradual improvement in the better half of the population. This achieved as re-initialized or mutated population members move into the "protected" half of the population, and the median of the population moves toward better objective function values. Since the process of reinitialisation effectively constitutes a uniformly-distributed random search, from the theory of random search it may be conjectured that the algorithm will find global minimizers of a problem, if allowed sufficient iterations.

6.2.3 Case Studies

A number of case studies drawn from interesting and challenging scientific and engineering applications were used to test and assess the performance of the individual algorithms. Generally, the case studies fall into 2 sets:

1. Smooth, with a dominant global minimum (Laser 1, Crack 1, Aerofoil - Figures 6.14(a), 6.14(c) & 6.14(e), and Rosenbrock's function)
2. Multiple local minima, non-convex (Laser 2, Crack 2, Bead - Figures 6.14(b), 6.14(d) & 6.14(f))

Laser 1 and 2

A two-dimensional test surface was derived from the computation of a quantum electrodynamical simulation of a laser-atom interaction experiment. The base case, Laser 1, is quite a smooth surface, the dataset containing only 4 minima, of which the global minimum is quite dominant, as can be seen in Figure 6.10(a). Additive fractal noise was overlaid on this dataset to develop a "noisier", more challenging surface to test the algorithms. This dataset, Laser 2, contained 1157 local minima of varying severity, and is illustrated in Figure 6.10(b).

Crack 1 and 2

Finite element analysis of a thin plate under cyclic loading, with a cutout specified by parameters, was used to generate the Crack datasets. Common practice in

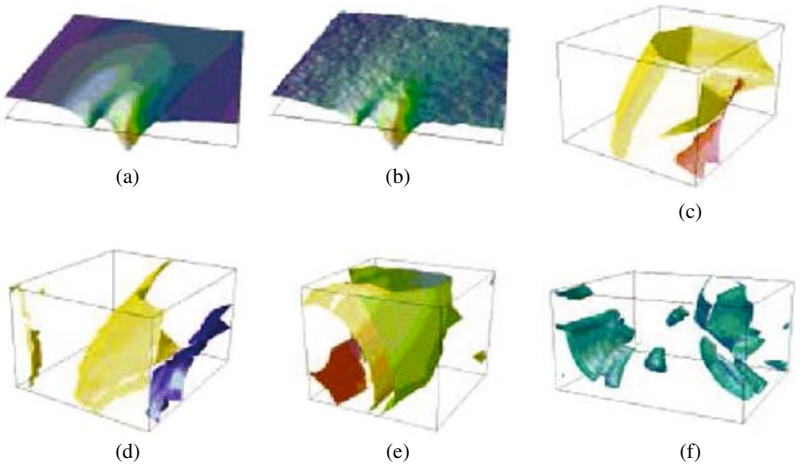


Fig. 6.10 Test case isosurfaces

damage tolerant design has been to minimise the maximum stress under load. Isosurfaces of these stress values are shown in Figure 6.10(c). This dataset, Crack 1, was reasonably smooth, with only 26 local minima. A new approach in modeling stressed components is to attempt to maximize durability. The finite element model of the crack-stress test case included fatigue cracks at a number of locations, and the objective of this test case, Crack 2, was to maximise the life of the part as determined by the time taken for fatigue crack growth to a defined length. Isosurfaces at a number of values are shown in Figure 6.10(d). In contrast to Crack 1, this dataset was "noisy", with 540 local maxima, and discontinuous isosurfaces.

Aerofoil

This test case models the aerodynamic properties of a two dimensional aerofoil. The objective function to be minimised is the lift-drag ratio, and this is computed by executing a Computational Fluid Dynamic model of the object. Figure 6.10(e) shows a number of isosurfaces in the parameter space investigated. The dataset was generally smooth, with only 12 local minima and a dominant global minimum.

Bead

The application from which this case study was drawn used a ceramic bead to minimise distortion of the radiation pattern of a mobile telecommunications handset during testing. The objective function value, derived from an FDTD full-wave analysis of the cable structure, was a measure of transmission strength through the bead at 1 GHz. The dataset for the Bead case study, of which isosurfaces for a particular value are shown in Figure 6.10(f), is quite complex and contains 298 local minima.

Rosenbrock's Function

In order to provide a point of comparison, the well-known Rosenbrock's function in two dimensions was included. The objective function values for this test case were directly computed from $f(x) = 100(x_2 - x_1)^2 + (-1 - x_1)^2$ for $x_i \in [-2, 2]$ which has one local minimum at $f(1;\ 1) = 0$.

6.2.4 Results of Numerical Experiments

For purposes of comparison a set of algorithms were tested, including EPSOC, a GA (Genesis 5.0), a parallel gradient descent method (P-BFGS), Dennis and Torczon's MDS, the Simplex method of Nelder and Mead, a variant of Simplex operating concurrently on multiple vertices with a search direction defined by a reducing set of vertices (RSCS), and variants of Simplex and RSCS utilizing line searching. All were parallel implementations; no serial algorithms were tested. The population-based algorithms, EPSOC and Genesis, used a population of 64 members. Other operational parameters of the algorithms, such as mutation and crossover rates for the GA, and extinction rate and mutation range for EPSOC, were tuned for each test case and the best results obtained are reported in this application. The extinction rate, nbad, varied between 5% and 15% of the population. Both algorithms are ideally suited to a master-slave, gather-scatter parallel system.

Each of the algorithms tested was run on each of the test cases from 10 randomly distributed start points. To allow direct comparison, in a given test case the same set of start points were used for each algorithm. The starting simplices for the direct search methods were right simplices aligned with the coordinate axes. By default they were reasonably large, scaled to 10% of the parameter range for each coordinate. The convergence criterion for most cases was a fractional step-wise gradient of 10^{-3}, except for EPSOC and Genesis, which used fixed limits on iteration count and function evaluations, respectively.

Table 6.3 shows the best objective function value obtained in 10 runs for each algorithm on each test case, and the Equivalent Serial Function Evaluations (ESFE) required to obtain that result. ESFE, derived from the number of steps involving concurrent evaluation of objective functions, is a measure of the actual time taken to find an optimal solution. For each test case, the best objective function value obtained by any algorithm, and the fastest ESFE to obtain that value, are highlighted. The ESFE results shown for EPSOC and the GA are the generation counts at which the minimum result was obtained, not the count for termination, which was fixed.

It may be noted from Table 6.3 that EPSOC effectively found the global minimum in 6 out of 7 cases (counting the result on the Crack 2 test case as successful, since it was within 0.03% of the global minimum value). In half of these cases, it also achieved this result faster than other algorithms. Its performance on Rosenbrock's function was equivalent to that of the other algorithms. The Kruskal-Wallis H test statistic was used to rank algorithms on each test case, based on all results returned. The Mann-Whitney U test was used for pairwise comparisons of the two highest-ranked algorithms, one of which was EPSOC on every test case. On 4 test

Table 6.3 Best objective functions values obtained in 10 runs, and time taken

Algorithm	Laser 1		Laser 2		Crack 1		Crack 2		Aerofoil		Bead		Rosenbrock	
	Obj.	ESFE	Obj.	ESFE	Obj.	ESFE	Obj.	ESFE	Obj.	ESFE	Obj.	ESFE	Obj.	ESFE
EPSOC	−0.48	10	−0.56	12	187.5	5	5356	20	−68.64	20	−39.85	14	1e-3	14
GA	−0.48	14	−0.56	19	188.5	16	5346	11	−68.51	9	−39.85	7	2e-4	23
P-BFGS	−0.48	37	−0.56	39	187.5	26	5347	12	−67.90	26	−29.71	18	7e-2	10
Simplex	−0.48	24	−0.56	24	187.6	25	5353	23	−68.64	20	−26.98	16	0	54
SimplexL1	−0.48	15	−0.56	24	187.5	13	5331	12	−68.63	18	−39.85	13	5e-6	48
MDS	−0.48	16	−0.56	12	187.6	13	5357	1000	−68.64	14	−16.12	7	3e-4	1000
RSCS	−0.48	22	−0.56	12	187.6	9	5347	42	−68.64	7	−26.91	12	0	39
RSCSL	−0.48	22	−0.56	24	187.5	15	5357	27	−67.21	18	−26.98	11	6e-6	88
RSCSL1	−0.48	13	−0.56	17	187.7	11	5348	8	−68.62	15	−26.98	5	8e-4	33

cases EPSOC was better than the second-ranked algorithm to a significance level of 0.05. On the remaining cases there was no statistically significant difference. In general, EPSOC achieved results as good as, or better, than all other algorithms, and its rate of convergence was highly competitive.

6.2.5 Conclusion

In this application a simple new Evolutionary Programming algorithm that utilizes concepts of Self-Organised Criticality has been described. Tested on a range of cases drawn from real-world problems, against a representative set of direct search, gradient descent and genetic algorithms, it has been demonstrated to exhibit superior performance. Examination of the results from the numerical experiments demonstrates population-based methods, EPSOC and the GA, are competitive against classical gradient descent and direct search methods providing they are executed on a parallel machine with sufficient processors to evaluate all of the population members in one iteration.. The quality of results obtained, and their speed in achieving them were clearly better in a majority of cases. Within the population-based methods, EPSOC outperformed the GA in finding the global minimum on all but one test case. Even for that case, the median result across multiple runs for EPSOC was better than that of the GA.

Like the GA, EPSOC is highly parallel, evaluating 1280 trial solutions in the same time it took Simplex, for example, to evaluate 81, and to generally better effect. The parallelism and completion time of the algorithm are set independent of problem size. The resulting pre-determined execution behaviour, in terms of resources required and time taken, its simplicity and its easily implemented master-slave parallelism make it well-suited for practical application to real world problems.

Given the intended use of these algorithms in practical engineering design, they have all, with the exception of Genesis 5.0, been integrated into the Nimrod/O framework. Nimrod/O is a design optimization toolset. It makes it possible to describe a design scenario using a simple, declarative language, then use parallel or distributed computers seamlessly to execute the experiment. Unlike other systems Nimrod/O combines optimization, distributed computing and rapid prototyping in a single tool. Further work is required on EPSOC to determine the efficacy of the method on problems of high dimensionality, and the link between problem dimensionality and optimal population size, extinction rates and mutation factors.

6.3 Evolutionary Computing as a Tool for Grammar Development

6.3.1 Introduction

An important trend in the field of Machine Learning sees researchers employing combinatory methods to improve the classification accuracies of their algorithms.

Natural language problems in particular benefit from combining classifiers to deal with the large datasets and expansive arrays of features that are paramount in describing this difficult and disparate domain that typically features a considerable amount of sub-regularities and exceptions. Not only system combination and cascaded classifiers are well established methods in the field of Machine Learning for natural language, also the techniques of bagging and boosting have been used successfully on a number of natural language classification task. These techniques hold in common that in no way do they alter the actual content of the information source of the predictor. Simply by re-distributing the data, different resamplings of the same classifier are generated to create a combination of classifiers. The field of evolutionary computing has been applying problem-solving techniques that are similar in intent to the aforementioned Machine Learning recombination methods.

Most evolutionary computing approaches hold in common that they try and find a solution to a particular problem, by recombining and mutating individuals in a society of possible solutions. This provides an attractive technique for problems involving large, complicated and non-linearly divisible search spaces. The evolutionary computing paradigm has however always seemed reluctant to deal with issues of natural language syntax. The fact that syntax is in essence a recursive, non-propositional system, dealing with complex issues such as long-distance dependencies and constraints, has made it difficult to incorporate it in typically propositional evolutionary systems such as genetic algorithms. Most GA syntactic research so far has focused on non-linguistic data, with some notable exceptions. Yet none of these systems are suited to a generic grammar optimization task, mainly because the grammatical formalism and evolutionary processes underlying these systems are designed to fit a particular task, such as information retrieval. Yet so far, little or no progress has been achieved in evaluating evolutionary computing as a tool for the induction or optimization of data-driven parsing techniques.

The GRAEL (**GRA**mmar **E**vo**L**ution) framework attempts to combine the sensibilities of the recombination machine learning methods and the attractive evolutionary properties of the concepts of genetic programming. It provides a suitable framework for the induction and optimization of any type of grammar for natural language in an evolutionary setting. In this application a general overview of GRAEL as a natural language grammar development technique is given.

6.3.2 Natural Language Grammar Development

Syntactic processing has always been deemed to be paramount to a wide range of applications, such as machine translation, information retrieval, speech recognition and the like. It is therefore not surprising that natural language syntax has always been one of the most active research areas in the field of language technology. All of the typical pitfalls in language like ambiguity, recursion and long-distance

dependencies, are prominent problems in describing syntax in a computational context. Historically, most computational systems for syntactic *parsing*, employ hand-written grammars, consisting of a laboriously crafted set of grammar rules to apply syntactic structure to a sentence. But in recent years, a lot of research efforts are trying to automatically induce workable grammars from annotated corpora, i.e. large collections of pre-parsed sentences. Since the tree-structures in these annotated corpus already implicitly contain a grammar, it is a relatively trivial task to induce a large-scale grammar and parser that is able to acquire reasonably high parsing accuracies on a held-out set of data.

Yet, data-analysis of the output generated by these parsers still brings to light fundamental limitations to these corpus-based methods. Even though they generally provide a much broader coverage as well as higher accuracy than hand-built grammars, corpus-induced grammars will still not hold enough grammatical information to provide structures for a large number of sentences in language, as some rules that are needed to generate the correct tree-structures are not induced from the original corpus. But even if there were such a thing as a full-coverage corpus-induced grammar, performance would still be limited by the probabilistic weights attributed to its rules. The GRAEL system described in this section tries to alleviate the problems inherent to corpus-induced grammars, by establishing a distributed evolutionary computing method for grammar induction and optimization. Generally, GRAEL can be considered as a system that allows for the simultaneous development of a range of alternative solutions to a grammatical problem, optimized in a series of practical interactions in a society of agents controlled by evolutionary parameters.

6.3.3 Grammar Evolution

A typical GRAEL society consists of a population of agents in a virtual environment, each of which holds a number of structures that allow them to generate sentences as well as analyze other agents' sentences. These grammars are updated through an extended series of inter-agent interactions, using a form of error-driven learning. The evolutionary parameters are able to define the content and quality of the grammars that are being developed over time, by imposing fitness functions on the society. By embedding the grammars in autonomous agents, grael ensures that the grammar development is grounded in the practical task of parsing itself.

The grammatical knowledge of the agents is typically bootstrapped by using an annotated natural language corpus. At the onset of such a corpus-based grael society, the syntactic structures of the corpus are randomly distributed over the agents, so that each agent holds a number of tree-structures in memory. The actual communication between agents is implemented in *language games*: an agent (**ag1**) presents a sentence to another agent (**ag2**). If **ag2** is able to correctly analyze **ag1**'s sentence, the communication is successful. If on the other hand, **ag2** is lacking the proper

grammatical information to parse the sentence correctly, **ag1** shares the necessary information for **ag2** to arrive at the proper solution.

A Toy Example

An example of a very basic language game is given here: Figure 6.11 shows a typical interaction between two agents. In this example, an annotated corpus of two sentences has been distributed over two agents. The two agents engage in a language game, in which **ag1** provides an assignment to **ag2**: **ag1** presents the sentence "I offered some bear hugs" to **ag2** for parsing. **ag2**'s knowledge does not contain the proper grammatical information to interpret this sentence the way **ag1** intended and so **ag2** will return an incorrect parse, albeit consistent with its own grammar.

ag1 will consequently try and help **ag2** out by revealing the minimal correct substructure of the correct parse that should enable **ag2** to arrive at the correct solution. **ag2** will incorporate this information in its grammar and try to parse the sentence again with the updated knowledge. Once **ag2** is able to provide the correct analysis (or is not able to after a certain number of attempts) either **ag1**'s next sentence will be parsed, or two other agents in the grael society will be randomly selected to play a language game.

Generations

This type of interaction is based on a high amount of knowledge sharing between agents. This extends the agents' grammars very fast, so that their datasets can grow very large in a short period of time. It is therefore beneficial to introduce new generations in the grael society from time to time. This not only allows for tractable computational processing times, but also allows the society to purge itself of bad agents and build new generations of good parser agents, who contain a fortuitous distribution of grammatical knowledge.

This introduces a neo-darwinist aspect in the system and involves the use of fitness functions that can distinguish good agents from bad ones. Typically the fitness of an agent is defined in terms of its parsing accuracy (i.e. the number of correct analyses), but the agents can be required to have fast and efficient grammars and the like. The use of fitness functions and generations ideally makes sure that the required type of grammatical knowledge is retained throughout different generations, while useless grammatical knowledge can be marginalized over time.

6.3.4 GRAEL-1: Probabilistic Grammar Optimization

GRAEL-1 is the most straightforward instantiation of grael and deals with probabilistic grammar optimization. Developing a corpus-based parser requires inducing

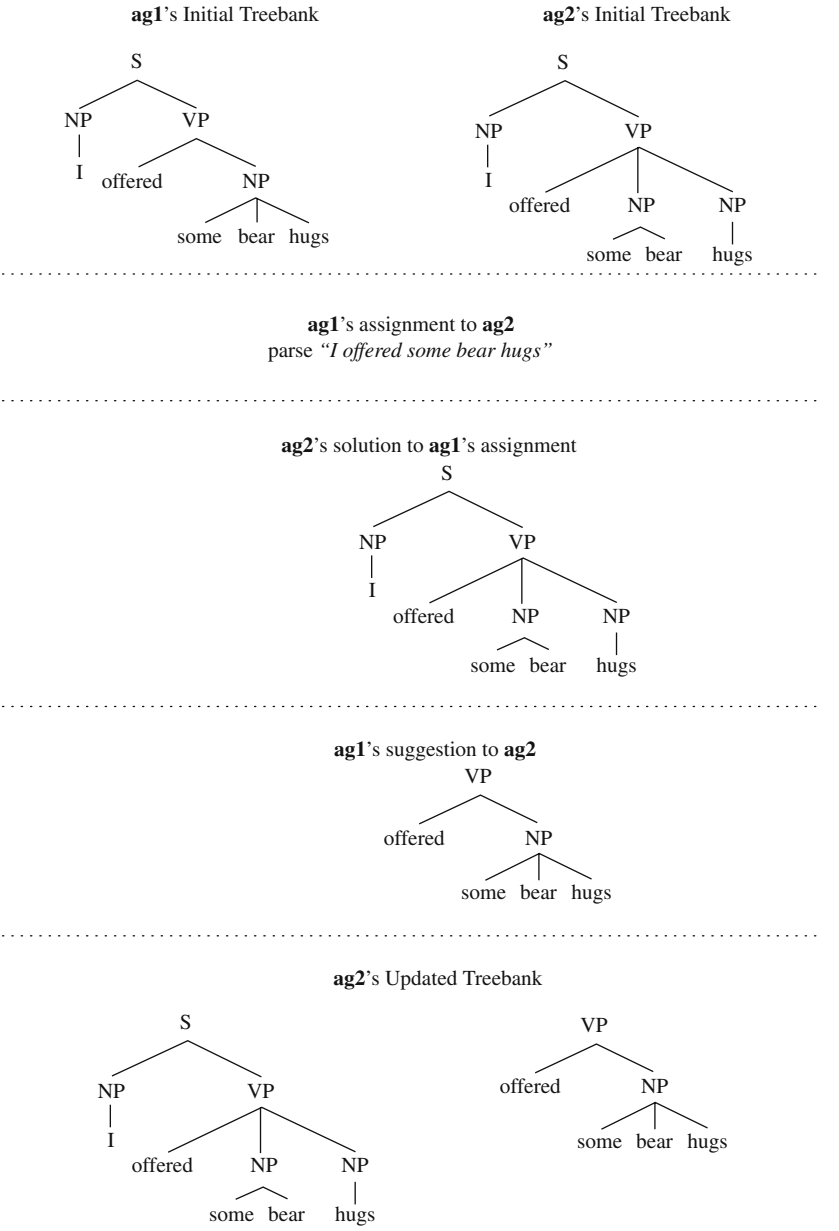


Fig. 6.11 A GRAEL language game

a grammar from an annotated corpus and using it to parse new sentences. Typically, these grammars are very large, so that for any given sentence a huge amount of possible parses is generated (a *parse forest*), only one of which provides the

correct analysis for the sentence. Parsing can therefore be considered as a two-step process: first a parser generates all possible parses for a sentence, after which a disambiguation step makes sure the correct analysis is retrieved from the parse forest. Fortunately, probabilistic information can also be induced from the annotated corpus2 that provides away to rank the analyses in the parse forests in order of probabilistic preference. Even though these statistics go a long way in providing well ordered parse forests, it can be observed in many cases that the ranking of the parse forest is sometimes counter-intuitive in that correct constructs are often overtaken by obviously erroneous, but highly frequent structures.

With GRAEL-1, an agent-based evolutionary computing method is proposed to resolve the issue of suboptimal probability mass distribution: by distributing the knowledge over a group of agents and having them interact with each other, multiple-route model is basically created for probabilistic grammar optimization. Grammatical structures extracted from the training corpus, will be present in different quantities and variations throughout the grael society (similarly to the aforementioned machine learning method of *bagging*). While the agents interact with each other and in effect practice the task on each other's grammar, a varied range of probabilistic grammars are optimized in a situation that directly relates to the task at hand (similarly to the machine learning method of *boosting*).

6.3.4.1 Experimental Setup

In the grael-1 experiments, the baseline accuracy is measured by directly inducing a grammar from the training set to power a parser, which disambiguates the test set. The same training set is then randomly distributed over a number of agents in the GRAEL society, who will consequently engage in a number of *language games*. At some point, the society is halted and the fittest agent is selected from the society. This agent effectively constitutes a redistributed and probabilistically optimized grammar, which can be used to power another parser. GRAEL-1 accuracy is achieved by having this parser disambiguate the same test set as the baseline parser.

6.3.4.2 Data and Tools

Two data sets from the Penn Treebank were used. The main batch of experiments was conducted on an edited version of the small, homogeneous atis-corpus, which consists of a collection of annotated sentences recorded by a spoken dialogue system. The larger Wall Street Journal Corpus (henceforth WSJ), a collection of annotated newspaper articles, was used to test the system on a larger scale corpus.

The parsing system pmpg is used, which combines a CKY parser and a postparsing parse forest reranking scheme that employs probabilistic information as well as a memory-based operator that ensures that larger syntactic contexts are considered during parsing.

Apart from different corpora, the experiment was conducted on different society sizes (5, 10, 20, 50, 100 agents), generation methods, fitness functions and methods to determine when to halt a grael society. New generations are created as follows: if an agent is observed not to acquire any more rules over the course of *n* communicative attempts3, it is considered to be an end-of-life agent. As soon as two end-of-life agents are available that belong to the 50% fittest agents in the society, they are allowed to procreate by crossing over the grammars they have acquired during their lifespan. This operation yields three new agents, two of which will take their ancestors' slots in the society, while the other one takes the slot of the oldest agent among the 50% unfit agents at that point in time. The fitness of an agent is defined by recording a weighted average of the F-score (see below) during inter-agent communication and the F-score of the agent's parser on a held-out validation set. This information was also used to try and halt the society at a global maximum and to select the fittest agent from the society. For computational reasons, the experiments on the WSJ-corpus were limited to two different population sizes (50 and 100) and used an approximation of GRAEL that can deal with large datasets in a reasonable amount of time.

Note that GRAEL-1 includes two different notions of crossover, although either one is a far stretch from the classic GA-type definition of the concept. The first type of crossover occurs during the language game when parts of syntactic tree-structures are being shared between agents. This operation relates to the recombination of knowledge aspect of crossover. The second type of crossover occurs when new agents are created by crossing over the grammars of two end-of-life agents. Again, the aspect of recombination is apparent in this operation. Note however the distinction with crossover in the context of genetic algorithms in that neither crossover operation in the GRAEL system occurs on the level of the genotype and that it is in fact the phenotype that is being adapted and which evolves over time.

6.3.4.3 Results

Table 6.4 displays the results of these experiments. Exact Match Accuracy expresses the percentage of sentences that were parsed completely correct, while the F-score is a measure of how well the parsers work on a constituent level. The baseline model

Table 6.4 Baseline vs. GRAEL-1 results

	ATIS		WSJ	
	Exact Match	$F_{\beta=1}$-score	Exact Match	$F_{\beta=1}$-score
Baseline	70.7	89.3	16.0	80.5
GRAEL (5)	72.4	90.9	—	—
GRAEL (10)	77.6	92.1	—	—
GRAEL (20)	77.6	92.1	—	—
GRAEL (50)	75.9	92.2	22.2	80.7
GRAEL (100)	75.9	92.0	22.8	81.1

is a standard pmpg parser using a grammar directly induced from the training set. Table 6.4 also displays scores of the GRAEL system for different population sizes. A significant gain is noticed for all GRAEL models over the baseline model on the atis corpus. The small society of 5 agents achieves only a very limited improvement over the baseline method. Data analysis showed that the best moment to halt the society and select the fittest agent from the society is a relatively brief period right before actual convergence sets and grammars throughout the society are starting to resemble each other more closely. The size of the society seems to be the determining factor controlling the duration of this period. In smaller societies, it may occur that convergence sets in too fast, since there is a narrower spread of data throughout the society. This causes convergence to set in prematurely, before the halting procedures even had a chance to register a proper halting point for the society. This leads to the low accuracy for the 5 agent society on the atis corpus.

Some preliminary experiments on a subset of the WSJ corpus had shown those society sizes of 20 agents and less to be unsuitable for a large-scale corpus, again ending up in a harmful premature stagnation. The gain achieved by the GRAEL society is less spectacular than on the atis corpus, but it is still statistically significant. Larger society sizes and full GRAEL processing on the WSJ corpus should achieve a larger gain, but is not currently feasible due to computational constraints.

The results show that grael-1 is indeed an interesting method for probabilistic grammar redistribution and optimization. Data analysis shows that many of the counterintuitive parse forest orderings that were apparent in the baseline model, are being resolved after grael-1 processing. It is also interesting to point out that an error reduction rate of more than 26% over the baseline method is achieved, without introducing any new grammatical information in the society, but solely by redistributing what is already there.

6.3.5 GRAEL-2: Grammar Rule Discovery

Any type of grammar, be it corpus-induced or hand-written will not be able to cover all sentences of language. Some sentences will indeed require a rule that is not available in the grammar. Even for a large corpus such as the WSJ, missing grammar rules provide a serious accuracy bottleneck. The grael-2 system described in this section provides a guidance mechanism to grammar rule discovery. In grael-2, the original grammar is distributed among a group of agents, who can randomly *mutate* the grammatical structures they hold. The new grammatical information they create is tried and tested by interacting with each other. The neo-darwinist aspect of this evolutionary system tries to retain any useful mutated grammatical information throughout the population, while noise is filtered out over time. This method provides a way to create new grammatical structures previously unavailable in the corpus, while at the same time evaluating them in a practical context, without the need for an external information source.

Some minor alterations need to be made to the initial grael-1 system to accomplish this, most notably the addition of an element of mutation. This occurs in the context of a language game at the point where **ag1** suggests the minimal correct substructure to **ag2**. In grael-1 this step introduced a form of error-driven learning, making sure that the probabilistic value of this grammatical structure is increased. The functionality of grael-2 however is different: It is assumed that there is a virtual noisy channel between **ag1** and **ag2** which may cause **ag2** to misunderstand **ag1**'s structure.

Small mutations on different levels of the substructure may occur, such as the deletion, addition and replacement of nodes in the tree-structure. This mutation introduces previously unseen grammatical data in the GRAEL society, some of which will be useless (and will hopefully disappear over time), some of which will actually constitute good grammar rules. Note again that the concept of mutation in the GRAEL system stretches the classic GA-notion. Mutation in grael-2 does not occur on the level of the genotype at all.

It is the actual grammatical information that is being mutated and consequently communicated throughout the society. This provides a significant speed-up of grammatical evolution over time, as well as enables a transparent insight into the grammar rule discovery mechanism itself.

6.3.5.1 Experimental Setup and Results

The grael-2 experiments have a similar setup to the GRAEL-1 experiments. For the experiments on the atis corpus, a special worst-case scenario test set was compiled to specifically test the grammar-rule discovery capabilities of GRAEL-2. This test set consists of 97 sentences that require a grammar rule that cannot be induced from the training set.

For the WSJ-experiments the standard test set was used. A 20-agent and a 100-agent society were respectively used for the ATIS and WSJ experiments.

Table 6.5 compares the grael-2 results to the baseline and grael-1 systems. The latter systems trivially achieve an exact match accuracy of 0%on the ATIS test set, which also has a negative effect on the F-score (Table 6.5). Grael-2 is indeed able to improve on this significantly. The results on the WSJ corpus show however that grael-2 has lost the beneficial probabilistic optimization effect that was paramount

Table 6.5 Baseline vs grael-1 vs grael-2 vs grael2+1 results

	ATIS		WSJ	
	$F_{\beta=1}$	Ex. Match	$F_{\beta=1}$	Ex. Match
Baseline	69.8	0	80.5	16
grael-1	73.8	0	81.4	22.8
grael-2	83.0	7.2	76.5	19.3
grael2+1	85.7	11.3	81.6	23.4

to grael-1. Another experiment was therefore conducted in which the grael-2 society is turned into a grael-1 society after the former's halting point. In other words: a society of agents is taken using mutated information and consequently grael-1 probabilistic redistribution is applied on these grammars. This achieves a significant improvement on all data sets and establishes an interesting grammar development technique that is able to extend and optimize any given grammar without the need for an external information source.

6.3.6 GRAEL-3: Unsupervised Grammar Induction

Grael-2 started off with an initial grammar that is induced from an annotated corpus, so that it could be considered to be a form of supervised grammar induction, since annotated data is required. Yet, annotated data is hard to come by and resources are necessarily limited. Recent research efforts however try to implement methods that can apply structure to raw data, simply on the basis of distributional properties and grammatical principles. Further alterations to the grael-2 system can extend its functionality to include this type of task.

The grael-3 system requires us to develop a basic, but workable grammar induction module that can build tree-structures on the basis of mutual information content calculated on bigrams. This module should not be considered a part of GRAEL-3 proper: any kind of grammar induction method can in principle be used to bootstrap structure in the society. Next, three types of experiment were performed for each data set: GRAEL-3a takes the whole training set and applies structure to those sentences based on information content values calculated on the entire data set. GRAEL-3b first distributes the sentences over the agents, after which the grammar induction module applies structure to these sentences based on information content values calculated for each agent individually. Since this grammar induction module effectively constitutes a parser, some experiments were also conducted in which parsing was only performed using this method (GRAEL-3ab-2).

The same training set/test set divisions were used as in the grael-1 experiments, while the experiments were performed on a 20-agent society for the atis-corpus and a 100-agent society for the WSJ-corpus. The F-score and the zero-crossing brackets was measured which is typically used to evaluate unsupervised grammar induction methods.

The first line of Table 6.6 shows the baseline accuracy using a PMPG on the training set annotated by the grammar induction module. These figures are very low, which is mainly due to the problematic labeling properties the grammar induction method imposes. A parser using ps-type rules such as PMPG does indeed need accurate node labels to be able to process grammatical structure in an accurate manner. The GRAEL-3a-1 system however is able to improve on this grammar significantly. Unsupervised grammar induction methods typically need a lot of data to achieve reasonable accuracy.

It is therefore not surprising that GRAEL-3b does not achieve any improvement over the baseline, since the grammar induction method only has a very limited amount of data to extract useful information content values from. Using the grammar induction method as a parser itself circumnavigates the problem of labeling and this has a positive effect on parsing accuracy. More importantly, GRAEL-3 seems again able to improve parsing accuracy significantly, both on the ATIS and the WSJ corpus.

6.3.7 Concluding Remarks

This section presented a broad overview of the GRAEL system, which can be used for different grammar optimization and induction tasks. Using the same architecture and only applying minor alterations, three different tasks were implemented: grael-1provides a beneficial re-distribution of the probability mass of a probabilistic grammar by using a form of error-driven learning in the context of interactions between autonomous agents. By introducing an element of mutation, GRAEL-1's functionality is extended and GRAEL-2 is projected as a workable grammar rule discovery method, significantly improving grammatical coverage on corpus-induced grammars. Following up GRAEL-2's grammar rule discovery method with grael-1's probabilistic grammar optimization proved to be an interesting optimization toolkit for corpus-induced grammars. Finally, GRAEL-3 is described as a first attempt to provide an unsupervised grammar induction technique. Even though the scores achieved by GRAEL-3 are rather modest compared to supervised approaches, the experiments show that the GRAEL environment is again able to take a collection of deficient grammars and turn them into better grammars through an extended process of inter-agent interaction.

The GRAEL framework provides an agent-based evolutionary computing approach to natural language grammar optimization and induction. It integrates the sensibilities of combinatory machine learning methods such as bagging and boosting, with the dynamics of evolutionary computing and agent-based processing. It is shown that GRAEL-1 and GRAEL-2 are able to take a collection of annotated data, providing an already well-balanced grammar, and squeeze more performance

Table 6.6 GRAEL-3 results

	ATIS		WSJ	
	$F_{\beta=1}$	0CB	$F_{\beta=1}$	0CB
Baseline (PMPG)	22.4	22.9	–	–
GRAEL-3A-I	25.6	24.9	–	–
GRAEL-3B-I	22.7	22.9	31.8	32.8
Baseline (GIM)	28.4	30.8	32.2	32.5
GRAEL-3AB-2	31.0	31.1	33.8	34.0

out of them without using an external information source. The experiments with grael-3 showed however that is equally able to improve on a collection of poor initial grammars, proving that the GRAEL framework is indeed able to provide an optimization for any type of grammar, regardless of its initial quality. This projects GRAEL as an interesting workbench for natural language grammar development, both for supervised, as well as unsupervised grammar optimization and induction tasks.

6.4 Waveform Synthesis using Evolutionary Computation

6.4.1 Introduction

Evolutionary Computation is being increasingly used in Computer Graphics to create scenes and animations. In these applications the rules are learned by the system through its interaction with the user. This property is used here to generate sound pattern variants. Other applications for interactive composition, using MIDI data to control music events with a heuristic approach are already studied.

From an algorithmic point of view, ESSynth can be described as a man-machine process that handles a set of rules to generate sound material. The main goal is to formulate a robust mathematical model for measuring waveform similarities and then defining genetic operations such as crossover and mutation to generate interesting sound transformations. By controlling the waveform target as well as the initial waveform population the user is able to obtain well-organized sounds or, at a higher level, to perform music composition. Such process can be understood as the next evolutionary step in the paradigm for musical composition. In each generation, the best waveform is sent to a sound output (a wavetable engine). ESSynth is an excellent tool for real-time synthesis where the user can manipulate the Target set as well as the initial population in order to get a musically significant sequence of waveforms and consequently new sounds. An external controller device may be linked to the ESSynth algorithm in order to control the production of sound patterns in real-time. ESSynth can be seen as an experimental environment for sound synthesis. This method is a kind of macro-structural synthesis so that the Fourier partials are treated in batches. From the operational point of view, it might be seen like carving tools that sculpt the set of waveforms.

6.4.2 Evolutionary Manipulation of Waveforms

The ESSynth method can be described as a man-machine interaction cycle. First, the user specifies a set of target waveforms. It is important to underline that the user is free to change the target set at any time. Second, the computer produces generations of waveforms using the target set as the raw material for the fitness

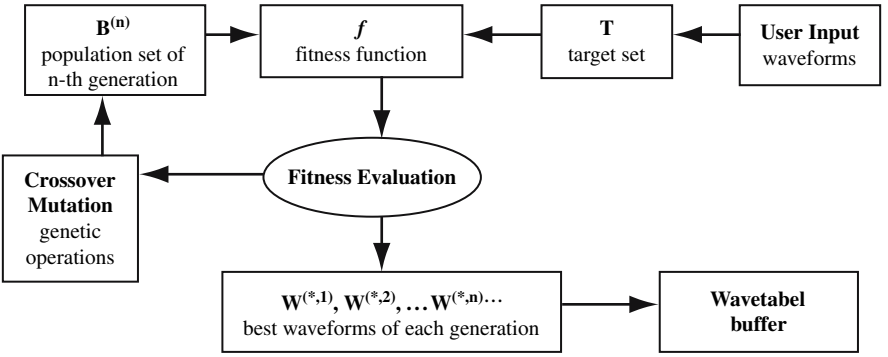


Fig. 6.12 ESSynth basic control structures

evaluation process. When the user changes the target, the computer generates a new set and so forth. There are three basic control structures; the population of the n-th generation denoted by $\mathbf{B}^{(\mathbf{n})}$ (the initial generation is denoted by $\mathbf{B}^{(\mathbf{0})}$), the set of target waveforms $\mathbf{T}$ and the Fitness Function $\boldsymbol{f}$. The fitness function is used to find the best waveform $\mathbf{w}^*$ of each generation $\mathbf{B}^{(\mathbf{n})}$. The $\mathbf{w}^*$ is defined as the most similar or, in mathematical terms, closest waveform $\mathbf{B}^{(\mathbf{n})}$ to the set $\mathbf{T}$, measured by a distance function (see the appendix for a definition). In each generation the best waveform $\mathbf{w}^*$ is sent to a buffer and is played as a periodically scanned wavetable. $\mathrm{W}^{(*,\mathrm{n})}$ denotes the best waveform of the n-th generation, which is sent to the output wavetable buffer.

Figure 6.12 shows the basic control structures where $\mathrm{w}^{(*,\mathrm{n})}$ denotes the best waveform of the nth generation, which is sent to the output wavetable buffer. The waveforms are normalized as floating-point arrays of 1024 values (or points) defined in the real interval $[-1, 1]$. The user always defines the $\mathbf{T}$ set and optionally the $\mathbf{B}^{(\mathbf{0})}$, which is otherwise randomly generated. When the user changes the $\mathbf{T}$, the system reacts in either of two ways depending upon its previous state: $\mathbf{B}^{(\mathbf{n})}$ is kept unchanged (as the initial $\mathbf{B}^{(\mathbf{0})}$) or a new $\mathbf{B}^{(\mathbf{0})}$ is randomly generated/user defined. The first situation produces a mutation in the waveform generation, but the overall characteristics of the sound pattern are kept unchanged. In the second case, the inputs will start a new generation of variant waveforms that will create new generations of sound patterns. The overlap-and-add technique is used to interlace the sequence of best waveform of each generation $\mathbf{w}^{(*,\mathbf{i})}$, in order to obtain a smooth audio output. To do so, a Hanning window is applied to each $\mathbf{w}^{(*,\mathbf{i})}$. The interlacing is done so that the amplitude doesn't change on the overlapped regions, as shown in Figure 6.13.

Crossover, Mutation and Fitness Evaluation

Waveform variants are produced by applying genetic operations such as crossover and mutation in the $\mathbf{B}^{(\mathbf{n})}$ population. ESSynth dynamically generates waveform

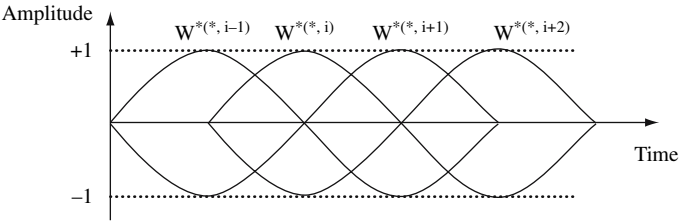


Fig. 6.13 Hanning windowed best waveforms $w^{(*,I)}$ overlap-and-add at 50%

sequences. Similar to the biologic evolution that produces diversity in Nature, it creates and manipulates a complex generation of sound material in real-time. In statistical terms the crossover increases the waveform co-variance and the mutation produces random variations in the population. These two operations are defined below.

Crossover

Similar to the genetic code of living organisms reproduced by meiosis, the crossover operation mixes the waveform codes (Figure 6.14) of population $\mathbf{B}^{(\mathbf{n})}$. In order to get better-fitted individuals, ESSynth crosses population $\mathbf{B}^{(\mathbf{n})}$ with the elements of the T set. This operation performs a natural selection in the domain of waveforms. As the user can interfere at any time, it would be more accurate to say that ESSynth uses a driven genetic selection.

In order to avoid glitch noise due to the crossover process, a convex combination and a Hanning window are applied to smooth the $\mathbf{S}^{(\mathbf{n},\mathbf{i})}$ waveform segment edges.

Mutation

In living organisms mutation implies a modification, usually due to external factors. It might affect individual phenotypes. This concept was used for the waveform mutation operation. A mutation parameter **b** is defined in the real interval [0,1] toquantify the mutation strength. When **b** is close to 0 the mutations are non-

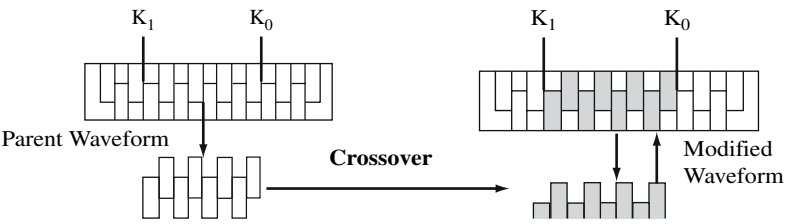


Fig. 6.14 Crossover operation diagram

existent; for **b** close to 1 the mutations are strong. This operation will add high frequency spectral components to the population of waveforms.

Fitness Evaluation

In the process of genetic improvement, it is necessary to measure which individuals are better fitted to survive. In Nature, an adverse environment selects these individuals. In the waveforms world this can be accomplished by a function that measures the distance between an n-th generation population set $\mathbf{B}^{(\mathbf{n})}$ and the target set **T**. Therefore it is necessary to have a function to measure the distance between two sets (population and target) not necessarily having the same number of elements (waveforms). In mathematics, this is called Hausdorff Distance. This is well suited for distance measurement between sets of points in a N-dimensional real space. In this case, N = 1024.

6.4.3 Conclusion and Results

A new methodology for sound generation was developed. Its mathematical model was established, constraining the sound variants in a coherent domain. ESSynth features include the use of a target set of waveforms, a fitness criterion using Hausdorff distance, an evolutionary process based on crossover and mutation operations (Figure 6.15). Table 6.7 shows the ESSynth parameters to generate a waveform sequence.

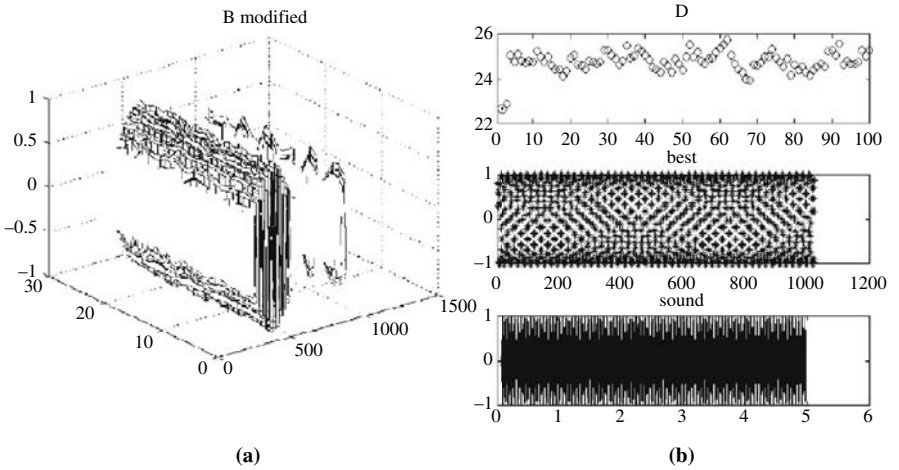


Fig. 6.15 a) Population of waveforms after 100 generations b) 1) Fitness variation of n-th generation population $B^{(n)}$ to T set; 2) Best (fitted) waveforms; 3) Resulting waveform after overlap and add

Table 6.7 ESSynth Table of Parameters to generate a waveform sequence

ESSynth Table of Parameters				
Index	Population Description	Frequency distribution (Hz)	Number of Individuals	Comments
01	Target set/harmonically distributed sine waves	100, 200, 300, … 2000	20	10% mutation
02	Population set/randomly distributed sine waves	From 180 to 16000	25	100 interactions

Appendix: Mathematical Model

An auxiliary metric, or distance function is firstly defined. This mathematical model considers waveforms as vectors in a real vector space $W = \Re^{1024}$, i.e. each vector in the space has 1024 components (points). Given two vectors **v** and **w** in **W**, the Euclidian Metric between them is:

$$d2(w,\ v) = (\Sigma i = 1, \ldots, 1024(wi - vi)^2)^{1/2} \tag{6.6}$$

As it is known, this metric induces the norm: $|w| = (\Sigma i = 1 \ldots 1024(wi)^2)^{1/2}$. Since these vectors are considered as waveforms, this norm gives the total energy of the resultant sound. Euclidian metric is arbitrarily used. Let $T = \{t^{(1)},\ t^{(2)}, .,\ t^{(L)}\}$ to be the Target Waveform Set and $B^{(n)} = \{w^{(n,1)},\ w^{(n,2)}, .,\ w^{(n,M)}\}$ the set of the n-th generation of the waveform population. Since these are subsets of W, the distance between them is defined as follows:

$$d(T,\ B^{(n)}) = \min\ \{d2(t^{(j)}, w^{(k)})\} \tag{6.7}$$

where: $j = 1, \ldots,\ L,\ k = 1, \ldots,\ M$
L is the number waveforms in the target T.
M is the number of waveforms in the n-th generation of the population $B^{(n)}$.

The measure (6.6) is called the "Hausdorff Distance" between two sets. As pointed above, T and $B^{(n)}$ are finite sets, therefore the minimum in Eq. 6.3 is given by at least one vector in $B^{(n)}$, which is denoted as $w^{(n,*)}$. This vector is the best waveform in the n-th generation of $B^{(n)}$ which means the closest waveform in population to the given target set T, that is using the metric defined in Eq. 6.7. Therefore, the Fitness Function of the n-th generation $f:\ T \times B^{(n)} \rightarrow B^{(n)}$ is

$$f(T,\ B^{(n)}) = w^{*(n)} \tag{6.8}$$

where T is the target set, $B^{(n)}$ is the n-th generation population, $w^{*(n)}$ is the best waveform.

Crossover Operation

Starting with a Crossover Vector described as $\alpha = [\alpha 1,\ \alpha 2, \ldots \alpha M]$ where $0 \leq \alpha i \leq 1$ chosen by the user, it is possible to define some sort of continuous waveform crossover. The best waveform $w^{(n,*)}$ of the n-th generation is used as, the Parent Waveform $w^{(n,*)} = (s_1,\ s_2,\ s_3, \ldots s_{1024})$ in $B^{(n)}$ and any other waveform in $B^{(n)}$ is denoted by $w^{(n,i)}$ where $0 \leq i \leq M$.

The following steps define the crossover operation in the n-th generation of the population:

1. Generate random integers in the interval [1, 1024].
2. Take two of these integers as $k1^{(n)}$ and $k2^{(n)}$, where $k1^{(n)} < k2^{(n)}$.
3. Select the segment of the waveform $w^{(n,*)}$ defined by k1 and k2 as $S^{(n,*)} = (s_{k1}, \ldots, s_{k2})$.
4. Combine the waveform segment $S^{(n,*)}$ with the correspondent segment of the waveform $Si^{(n,i)}$ in $B^{(n)}$
5. Apply a Hanning Window $H(Si^{(n,*)})$
6. Combine the windowed segment with the equivalent segment $S^{(n,i)}$ of $B^{(n)}$.

$$S^{(n+1,i)} = \alpha i.H(Si^{(n,*)}) + (1 - \alpha i)\,.\,S^{(n,i)} \tag{6.9}$$

where $S^{(n+1,i)} = (s'_{k1}, .., s'_{k2})$.is the new segment.

The crossover operation is the replacement of each $S^{(n+1,i)}$ in the original waveform making $w^{(n+1,i)} = (s_1,\ s_2,\ s_3, \ldots s'_1,\ s'_2,\ s_{1024})$

Repeat steps (4) and (5) for all waveform $w^{(n,i)}$ in $B^{(n)}$, so that $w^{(n,i)} \neq w^{(n,*)}$.

Observation: A Hanning window is applied in steps (5) and (6) a also a convex combination to fade the borders of the waveform segment $S^{(n,i)}$.

Mutation Operation

This operation starts with the definition of a Mutation Coefficient $0 \leq b \leq 1$ that fixes the amount of disturbance applied on $B^{(n)}$. Since the waveforms belong to $W = \Re^{1024}$, a 1024 points Mutation Vector β is randomly generated in the disturbance interval [1-b, 1]. The Mutation Operation is defined on the n-th generation by the following steps:

1. Create the Mutation Vector $\beta = [\beta 1,\ \beta 2,\ \beta 3 \ldots \beta 1024]$, where each βj belongs to the disturbance interval [1-b, 1].
2. On the elements of $B^{(n)} = \{w^{(1,n)},\ w^{(2,n)}, \ldots,\ w^{(M,n)}\}$ apply the operation of disturbance:

$$_w(j,\ n+1) =_w (j,n)_\beta$$

Repeat the steps (1) and (2) for every next generation.

6.5 Scheduling Earth Observing Satellites with Evolutionary Algorithms

6.5.1 Introduction

A growing fleet of NASA, commercial, and foreign Earth observing satellites (EOS) uses a variety of sensing technologies for scientific, mapping, defense and commercial activities. As the number of satellites (now around 60) increases, the system as a whole will begin to approximate a sensor web. Image collection for these satellites is planned and scheduled by a variety of techniques, but nearly always as separate satellites; not as an integrated sensor web. Since activities on different satellites, or even different instruments on the same satellite, are typically scheduled independently of one another, manual coordination of observations by communicating teams of mission planners is required. As sensor webs with large numbers of satellites and observation requests develop, manual coordination will no longer be possible. Schedulers that treat the entire web as a collection of resources will become necessary. Scheduling EOS is complicated by a number of important constraints. Potin lists some of these constraints as:

1) Power and thermal availability.
2) Limited imaging segments per orbit. In a given orbit, a satellite will pass over a target only once. For the sun-synchronous orbits used by most Earth observing satellites, each orbit takes about 90 minutes.
3) Revisit limitations. A target must be within sight of the satellite; and EOS satellites travel in fixed orbits. These orbits pass over any particular place on Earth at limited times so there are only a few imaging windows (and sometimes none) for a given target.
4) Time required to take each image. Most Earth observing satellites take a one dimensional image and use the spacecraft's orbital motion to sweep out the area to be imaged. Thus, the larger the image the more time is required to take it.
5) Limited on-board data storage. Images are stored in a solid-state recorder (SSR) until they can be sent to the ground.
6) Ground station and communication satellite availability, especially playback opportunities. The data in the SSR can be sent to the ground either when the satellite passes over a ground station or via geosynchronous communication satellites. Ground station windows are as limited as any other target, and suitable communication satellites (mostly TDRSS) are only available when not servicing higher priority flights (e.g., shuttle or station).
7) Transition time between look angles (slewing). Some instruments are mounted on motors that can point either side-to-side (cross-track) or forward and back (along track). In addition, some satellites can rotate to point their instruments in any direction. These are called agile satellites.
8) Cloud cover. Some sensors cannot see through clouds. Not only do clouds cover much of the Earth at any given time, but some locations are nearly always cloudy.

9) Stereo pair acquisition.
10) Coordination of multiple satellites. In a sensor web an imaging request can be satisfied by any of several satellites. Also, in many cases there is a need to image a particular area by more than one sensor, often with time constraints.

EA can effectively schedule earth-imaging satellites, both single satellites and cooperating fleets. The constraints on such fleets are complex and the bottlenecks are not always well understood, a condition where evolutionary algorithms are often more effective than traditional techniques. Traditional techniques often require a detailed understanding of the bottlenecks, whereas evolutionary programming requires only that one can represent solutions, modify solutions, and evaluate solution fitness, not actually understand how to reason about the problem or which direction to modify solutions (no gradient information is required, although it can be used).

Model Problems

Since this application is designed to consider the scheduling of a parameterizable generic system, not any particular spacecraft, sensor, or sensor web, it is important to develop a set of model problems that exhibit important aspects of EOS scheduling now and in the future. This application is attempted to base the model's sensors and satellites on hardware currently in orbit. The scope is limited to seven problems:

1) A single satellite with a single cross-track slewable instrument.
2) A two-satellite constellation with satellites identical to that in problem one.
3) A single agile satellite with one instrument.
4) A single satellite with multiple instruments (one of which is slewable).
5) A sensor web of single-and multiple-instrument satellites communicating directly with the ground.
6) A sensor web of single-instrument agile satellites communicating with an in-orbit communications system based on high-data-rate lasers.
7) A sensor web with a very large number of satellites including satellites with multiple instruments. This problem presumes much cheaper and more reliable launch.

Problems 1 and 2 have been implemented. The Results section compares a number of search techniques against problem 2 with the following characteristics:

1) One week of satellite operations. 2) Two satellites in sun synchronous orbit one minute apart. 3) One identical instrument per satellite. 4) Slewing up to 48 degrees cross-track in either direction at a rate of 50 seconds/degree for each instrument. 5) 4200 imaging targets (takeImages) randomly distributed around the globe; 123 of these never come into view of either satellite. 6) 24 seconds data recording per takeImage. 7) A priority between 1 and 5 (higher priority is more important) for each takeImage.

6.5.2 EOS Scheduling by Evolutionary Algorithms and other Optimization Techniques

There are a number of optimization (evolutionary and otherwise) algorithms in the literature. This application compares a genetic algorithm (GA), simulated annealing (SA), and stochastic hill climbing (HC). In addition, random and squeaky wheel (SW) transmission operators are compared. Random transmission operators change a schedule at random (consistent with the constraints). Squeaky wheel operators examine a schedule and try to make changes that are likely to improve the schedule.

The schedule is represented as a permutation (the genotype) of the image requests (takeImages). A simple, deterministic greedy scheduler assigns resources to the requested takeImages in the order indicated by the permutation. This produces a timeline (the phenotype) with all of the scheduled takeImages, the time they are executed, and the resources used. The greedy scheduler assigns times and resources to takeImages using earliest-first scheduling heuristics while maintaining consistency with sensor availability, onboard memory (SSR) and slewing constraints. If a takeImage cannot be scheduled without violating constraints created by scheduling takeImages from earlier in the permutation, the takeImage is left unscheduled.

Simple earliest-first scheduling starting at epoch $(\text{time} = 0)$ had some problems, and it was discovered that the algorithm works better if 'earliest-first' starts with a particular imaging window (period where the satellite is within sight of a target; most takeImages have several windows in the week-long problem) rather than at epoch. If the takeImage cannot be scheduled before the end of time, the algorithm starts at epoch and continues until the takeImage is scheduled or the initial imaging window is reached. The window within which a takeImage is scheduled is stored in memory and used by children when they generate schedules. The extra scheduling flexibility may explain why this approach works better than earliest-first starting at epoch.

Constraints are enforced by representing each resource as a timeline. Scheduling a takeImage causes each relevant resource timelines to take on appropriate values (i.e., in use for a sensor, slew motor setting, amount of SSR memory available) at different times. A takeImage is inserted at the first time examined and available in all the required resource timelines.

Search is guided by a fitness function that determines the 'goodness' of a schedule generated from a permutation. The fitness function must provide a fitness for any possible schedule, no matter how bad it is, and nearly always distinguish between any two schedules, no matter how close they are. The fitness function is multi-objective. The objectives include:

1) Minimize the sum of the priority of the images not scheduled (takeImages). Each takeImage has a priority between 1 and 5, where the larger numbers indicate higher priority.
2) Minimize total time spent slewing (slew motors wear out).
3) Minimize the sum of the slew angles for the images taken (small slews improve image resolution). These objectives are manipulated so that lower values are better fitness; the objectives are then combined into a weighted sum:

$$F = w_p \sum_{I_u} I_p + w_s S_t + w_a \sum_{I_s} I_a \tag{6.10}$$

where F is the fitness, I_{u} is the set of unscheduled takeImages, I_{s} is the set of scheduled takeImages, I_p is the priority of a takeImage, S_t is the total time spent slewing, Ia is the slewing angle the schedule requires for a takeImage, and w_p, w_s, and w_{a} are weights (positive numbers).

The three search algorithms are:

1) Stochastic hill climbing (HC) starts with a single randomly generated permutation. This permutation (the parent) is mutated to produce a new permutation (a child) which, if it produces a better (more fit) schedule than the parent, replaces the parent. Two cases are investigated: five restarts per run and no restarts. With no restarts, each search generates 100,000 children starting with a random permutation. In the restart case, each search consists of five sub-searches of 20,000 children each; the best individual from all five searches is reported.
2) Simulated annealing (SA) is similar to HC except less fit children can replace the parent with probability $p = \frac{-\Delta F}{T}$ where ΔF is how much less fit the child is. The temperature starts at 100 and is multiplied by 0.92 every 1000 children (100,000 children are generated per run).
3) The genetic algorithm (GA) seeks to mimic the natural evolution of populations of organisms and there are many variants. The GA employed in this application is as follows:

   a) Generate a population of 100 random permutations
   b) Calculate the fitness of each permutation
   c) Repeat
      i) Randomly select parent permutations with a bias towards better fitness
      ii) Produce child permutations from the parents with: A) crossover that combines parts of two parents into a child, or B) mutation that modifies a single parent
      iii) Calculate the fitness of the child.
      iv) Randomly replace individuals of less fitness in the population with the children.
   d) Until 100,000 children have been produced.

The search for a good schedule starts with one or more random permutation (the initial parents) and uses mutation and crossover operators to create children from parents. This application compares four mutation operators and one crossover operator. The mutation operators are: 1) Random swap. Two permutation locations are chosen at random and the takeImages are swapped. Swaps are executed 1–9 times per mutation. A single random swap is called order-based mutation. 2) Squeaky swap. This is the same as random swap except that the takeImages to swap are chosen more carefully. Specifically, a tournament of size 10, 20, 50, 100, 200, or 500 selects both takeImages. One takeImage that 'should' be moved forward in the permutation is chosen. The winning takeImage is (in this order): a) unscheduled rather than scheduled b) higher priority c) later in the permutation The other takeImage

is chosen assuming it should be moved back in the permutation. This tournament winner has the opposite characteristics. Although the takeImages to swap are chosen because one 'should' move forward in the permutation and the other 'should' move back, this is not enforced. Experiment determined that the desired direction of the swap did not actually occur nearly as often as expected, occasionally less than half the time! 3) Placed squeaky swap. Here the direction is enforced. A separate tournament (of size 10, 20, 50, 100, or 200) is conducted for each takeImage. The takeImage to move forward is forced to be in the last half of the permutation. The takeImage to move back is then forced to be at least half way towards the front. 4) Cut and rearrange. The permutation is cut into 1–5 pieces and these are put back together in a random order. This is similar to the cut-set based operators used in the traveling salesman problem community. The crossover operator is only used in the genetic algorithm. The operator is Syswerda and Palmucci's position-based crossover. Roughly half of the permutation positions are chosen at random (50% probability per position). These positions are copied from the father to the child. The remaining takeImage numbers fill in the other positions in the order they appear in the mother. In many cases several different transmission operators and/or the same kind of operator with different sized tournaments, number of swaps, or cuts were used. In these cases, each child was produced by a randomly chosen transmission operator.

6.5.3 Results

A number of search technique/transmission operator pairs were compared. Each combination was repeated 94 times to get statistically significant results. The resulting distributions were spot checked for a gaussian distribution to insure the Student's T-test is valid. In each trial, evolution produced 100,000 children. A quantitative comparison of search techniques and transmission operators (various forms of mutation and crossover) can be found in table I. The techniques at the top of the table produce the best schedules, the techniques at the bottom the worst. A few observations: 1) Simulated annealing is clearly the best search technique. It is not surprising the SA beats HC, since HC is clearly vulnerable to local minima. To understand why SA and HC beat GA, consider the building blocks in the permutation. These may be thought of as sets of takeImages in a particular order that leads to good partial schedules. Moving an arbitrary takeImage before a building block can easily disrupt it by making some of the takeImages unschedulable; or worse, causing one of the takeImages in the building block to be scheduled in another window further disrupting the building block. Since good building blocks are thought to be essential to GA performance, GA does poorly. 2) Random swap mutations beat the smarter 'squeaky' mutation where the takeImages to swap are chosen more carefully (a counter intuitive result). This may be, in part, because the squeaky operators limit the possible moves an algorithm may take. This can create additional local minima that the search then falls into. 3) Multiple swaps are better than a single

swap, possibly because some moves are impossible with a single swap. 4) Ordering techniques by priority or takeImage rather than fitness doesn't make any difference for the best techniques, and much of the difference that does occur is not statistically significant. 5) The cut and rearrange operators do very poorly. Cut and rearrange works well for the traveling salesman problem because moving contiguous chunks of the permutation relative to each other does not change the partial fitness of the chunk. In permutation driven scheduling, however, reversing the order of two contiguous chunks can cause very large changes in the schedule. These observations should be considered preliminary rather than definitive. First of all, this is a single problem and results may vary when a larger range of the model problems are addressed. Second, the squeaky algorithms can stand improvement and may someday outperform the random operators. Nonetheless, if these results stand up, there are some important implications.

1) Simulated annealing requires less memory than the genetic algorithm and does not require crossover operators or a population, making it better performing, more efficient, and easier to implement.
2) Random swaps out perform the 'smarter' squeaky swaps, making random swaps better performing, faster, and easier to implement.
3) One should allow multiple random swaps, in spite of the minor increase in code complexity.

Figure 6.16 shows the evolutionary history of the best individuals for the best schedules evolved by simulated annealing (SA), hill climbing (HC), and the genetic algorithm (GA) using the one random swap mutation operator. Notice that although simulated annealing wins in the end, it trails GA until about generation 50 and trails HC until about generation 70. SA seems to be doing a better job of finding and then exploiting a deep minimum. Notice also that all three techniques are still improving

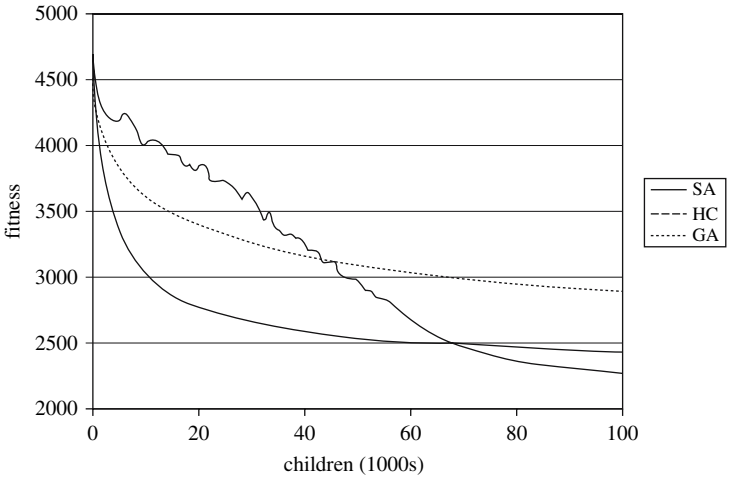


Fig. 6.16 A comparison of evolutionary history based on SA, HC and GA

the schedule at the end of the run, suggesting that additional evolution (more than 100,000 children) would be rewarded with better schedules.

One unexpected property of the schedules generated was the slewing. Specifically, in order to minimize total slewing time (St from equation 1) the schedules tended to place takeImages such that the instrument is slewed to extremes; which will generate relatively low resolution images. This could perhaps be improved if the fitness function gave more weight to minimizing the sum of the slews or if the instruments slewed faster (which would also be more realistic).

A second experiment compared GA with one swap operator on two problems: in the first, each satellite was randomly assigned half of 4,200 takeImages; in the second, either satellite was allowed to execute any takeImage as seen in Table 6.8. As might be expected, the case where any satellite could take any image produced superior schedules. Specifically, the shared case was able to take about 28% more images, the priority measure improved 40%, and fitness 35%. This suggests that integrated fleet scheduling is much better than separately scheduling each satellite or sensor.

6.5.4 Future Work

Future work will be focused to add: 1) Additional model problems. 2) A duty cycle constraint. This constraint requires that an instrument is not used for more than u seconds in any t second time period. 3) Improved squeaky operators; in particular, shifting a high priority, unscheduled takeImage forward, rather than swapping with a scheduled, low priority takeImage. 4) Swap operators where the number of swaps is a probabilistic function of the number of children that have been produced. As evolution proceeds, the number of swaps is reduced. This encourages large steps in the beginning of evolution and smaller refinement steps near the end. 5) Transmission operator evolution; where transmission operators that have done well early in evolution are more likely to be used. 6) Additional forms of local search. 7) HBSS (Heuristic Biased Stochastic Search) with contention based heuristics.

Table 6.8 Comparison of shared target vs separate targets for a two satellite constellation using ga with only single swap mutation and Crossover. All comparisons are statistically significant. The shared case is 25–40% better depending on the measure used for Comparison

Problem	Fitness (equation 1)	Priority ($w_p \sum_{I_u} I_\mathrm{p}$)	TakeImage (I_u)
shared	2171	1873	1199
separate	3346	3096	1657

6.6 An Evolutionary Computation Approach to Scenario-based Risk-return Portfolio Optimization for General Risk Measures

6.6.1 Introduction

Due to the increasing need for financial decision optimization algorithms for complex and non-convex optimization problems in the area of financial engineering, the amount of available biologically inspired algorithms for financial modeling increased significantly during the last years.

In this application, the well-known stochastic single-stage scenario-based risk-return portfolio optimization problem is considered. To solve this problem, a standard genetic algorithm is used, which is summarized in Table 6.9. Evolutionary computation approaches have been successfully applied to this class of portfolio optimization problems.

An analysis of the proposed methods reveals that the main focus is laid on multi-criteria optimization and especially the restriction to the pure Markowitz approach, i.e. using solely the expectation vector and correlation matrix of the financial assets under consideration. To extend the application domain to general risk measures the stochastic scenario-based formulation of single-stage portfolio optimization is used. By using an evolutionary computation approach the same optimization technique and decision support framework for every loss distribution-based risk measure is applied, regardless its underlying structure.

Consider the example of Value at Risk (VaR). While VaR leads to a non-convex and non-differential risk-return optimization problem for general loss distributions, the evaluation of VaR given some specific loss distribution is simply its quantile.

The problem of estimating the correct correlation matrix, which is often mentioned and criticized, is also avoided by using the scenario-based approach.

6.6.2 Portfolio Optimization

In the classical bi-criteria portfolio optimization problem an investor has to choose from a set of (financial) assets A with finite cardinality $a = |A|$ to invest his available

Table 6.9 Meta-heuristic: evolutionary computation-genetic algorithm

1	$P \leftarrow$ *Generate Initial Population*
2	*Evaluate(P)*
3	while termination conditions not met do
4	$P' \leftarrow Recombine(P)$
5	$P'' \leftarrow Mutate(P')$
6	$Evaluate(P'')$
7	$P \leftarrow Select(P \cup P'')$
8	end while

budget. The bi-criteria problem stems from the fact that the investor aims at maximizing his return while aiming at minimizing the risk at the same time. The stochastic problem definition depends on the formulation of the underlying uncertainty. Instead of using an expectation vector of size a and a correlation matrix of size a x a like the standard Markowitz approach suggests, the stochastic scenario-based approach is applied, i.e. the decision is based on a set of scenarios S with finite cardinality $\mathrm{s} = |S|$ where each $\mathrm{s_i}$ is equipped with a probability pi with $\sum_{j=1}^{s} pj = 1$. Each scenario contains one possible development of all assets under consideration.

Let $x \in R^a$ be some portfolio, with budget normalization, i.e. $\sum_{a \in \mathscr{A}} x_a = 1$, i.e. each $\mathrm{x_i}$ denotes the fraction of budget to invest into the respective asset i. S is now rewritten as a matrix S to calculate the discrete Profit & Loss (P& L) distribution l for some portfolio x, which is simply the cross product $\ell_x = \langle x,\ S \rangle$. Let $x^*_\rho \in \mathbb{R}^a$ denote the optimal portfolio given some risk measure ρ and ℓ^*_ρ denote the respective ρ-optimal discrete loss distribution.

When the bi-criteria aspect of this portfolio optimization problem is reconsidered, the loss distribution may be subsequently mapped to these two dimensions the return (reward, value) dimension is the expectation $\mathrm{E}(l_x)$ and the risk dimension is the risk mapping $\rho(l_x) : \mathrm{R^s} \longrightarrow \mathrm{R}$. The risk dimension received special importance due to regulatory frameworks like Basel II, as well as the academic discussion about coherence of risk measures. In this notation, a risk measure is a statistical parameter of the loss distribution l_x, e.g. the Standard Deviation in the Markowitz case, the quantile in the VaR case, or the expectation of the quantile in the Conditional Value-at-Risk (CVaR, see below) case.

Furthermore, let the set x denote all organizational, regulatory and physical constraints. Basic constraints, which are commonly included into x are:

- upper and lower limits on asset weights: $\chi a \geq l, \chi a \leq u \forall a \in A$
- minimum expected profit: $\mathrm{E}(l_x) \geq \mu$
- maximum expected *risk*: $\rho(l_x) \leq \rho$

Often, the constraint of disallowing short selling is implicitly added, and would be modeled by setting $l = 0$ in this formulation. While such basic constraints above and their extensions can be reformulated as convex optimization problems, more involved constraints like e.g. cardinality constraints, or non-linear, non-differentiable (i.e. combination of fixed and non-linear flexible cost) transaction cost structures, as well as buy-in thresholds, or round lots lead to non-convex, non-differential models, and have motivated the application of various heuristics such as evolutionary computation techniques.

In the non-multi-criteria setting, three different main portfolio optimization formulations are commonly used. Either minimize the risk

$$\text{minimize x} : \rho(l_x), \text{ subject to } x \varepsilon X, \tag{6.11}$$

or maximize the value (expectation)

$$\text{maximize x} \; : \; \mathrm{E}(l_x), \; \text{subject to} \; x \varepsilon X, \tag{6.12}$$

or apply the classical bi-criteria optimization model, where an additional risk-aversion parameter κ is defined, i.e.

$$\text{maximize x} \; : \; \mathrm{E}(l_x) - \kappa\rho(l_x), \; \text{subject to} \; x \varepsilon X, \tag{6.13}$$

The third formulation () is used in this application, which is a direct reformulation of bi-criteria problem to a single-criteria problem. If formulation (6.11) or (6.12) is used, the respective constraint to limit the respectively other dimension has to be integrated to ensure the equivalence.

It should be noted, that the minimum expected profit constraint is necessary for calculating efficient frontiers, i.e. by iterating over a set of minimal expected profits, which are calculated from the scenario set S. The critical line can also be calculated conveniently by iterating over the parameter κ.

6.6.3 Evolutionary Portfolio Optimization

A genetic algorithm approach is adopted to this stochastic portfolio optimization problem. The first issue is the selection of the genotype structure. One may either use the real-valued phenotype-equivalent or use some sort of bit-encoding. For the scenario-based framework presented below, genotype-phenotype-equivalent representation is opted. It is mainly based on preliminary experiences with large scenario-sets with a huge amount of assets a, i.e. a ≥ 100. This necessitates the need for normalization after each mutation to ensure the basic budget constraint, i.e. $\sum x_{a \in A} = 1$. This can be done on the fly after each budget-constraint violating operation.

The effect of using different crossovers for the portfolio selection problem has already been investigated. Three types of crossovers have been compared in the previous application: discrete N-point crossovers with $\mathrm{N} = 3$, intermediate crossovers, as well as BLX-α crossovers. For the results below, both N-point crossovers and intermediate crossovers have been used, as those two have shown to suffice the needs for solving this optimization problem.

Table 6.10 displays the structure of the portfolio weight chromosome and the fitness calculation. For each chromosome (portfolio) x_c its respective loss distribution l_{x_c} is calculated, from which the expectation $\mathrm{E}(l_{x_c})$ and respective risk $\rho(l_{x_c})$ is computed. These two dimensions (return and risk) are mapped to a single value via a risk-aversion parameter κ, which is commonly set to 1. The aim is to maximize the fitness value, i.e. to maximize the expected return and simultaneously minimize the expected risk. The example below shows a 95% Value at Risk with an inversion of the sign, as a higher VaR equals lower risk.

The following functions have been used for the implementation of the evolutionary (genetic) algorithm:

– c = evoGArealInitial(n, z)-Generate random chromosome c of length n with a maximum of z zeros.

- c = evoGArealMutationFactor(c, f, p)-Mutate chromosome c by multiplying up to a maximum of p randomly chosen genes with factor f.
- c1, c2 = evoGArealCrossoverIntermediate(p1, p2)-Perform an intermediate crossover on parents $p1$ and $p2$ to generate children $c1$ and $c2$.
- c1, c2 = evoGArealCrossoverNpoint(p1, p2, n)-Perform an n-point crossover on parents $p1$ and $p2$ to generate children $c1$ and $c2$.
- evoGArealNormalization(c, delta) -Normalizes the chromosome c with $ci = \frac{ci}{\sum c}$. Each gene of c smaller than δ is set to 0 before normalization.

The special structure of the portfolio optimization problem is considered during the initial generation of random chromosomes in evoGArealInitial due to the modification of (up to) a pre-specified number of randomly chosen genes to zeros. Furthermore, the mutation evoGArealMutationFactor with $f = 0$ is explicitely used to create portfolio chromosomes, where some asset weights are zero. This is due to the fact, that optimal portfolios (applying commonly used risk measures) generally include a small subset of all assets under consideration.

6.6.4 Numerical Results

Implementation

The evolutionary algorithm framework has been implemented in MatLab 7.3.0 (Release 2006b). Besides the genetic algorithm functions described above, fitness selection procedures, various risk evaluations as well as an algorithm workflow engine has been developed. The code is freely available on the Web under an Academic Free License at http://www.compmath.net/ronald.hochreiter/

Data

The scenario set used for further calculations consists of daily historical data of 15 Dow Jones STOXX Supersector indices (Automobiles & Parts, Banks, Basic Resources, Chemicals, Construction & Materials, Financial Services, Food & Beverage, Health Care, Industrial Goods & Service, Insurance, Media, Oil & Gas, Technology, Telecommunications, Utilities). The chosen time-frame was January 2002 to January 2006 (i.e. 4 years of data resulting in 1005 scenarios of daily index changes).

Table 6.10 Example chromosome-fitness: κ-weighted return and risk

Asset 1 Asset 2… Asset a	Return $\mathbb{E}(l_x)$	Risk $\rho(l_x)$	Fitness $\mathbb{E}(l_x) - \kappa\rho(l_x)$
$x_c = (x_1\, x_2 \ldots x_a)$	$\mathbb{E}(l_{x_c})$	$\mathrm{VaR}_{0.95}(l_{x_c})$	$\mathbb{E}(l_{x_c}) + \mathrm{VaR}_{0.95}(l_{x_c})$

Risk Measures

Three of the most commonly applied risk measures ρ for portfolio management has been chosen to conduct a comparison, and to show the general applicability of the method presented in this application. Standard Deviation, Value at Risk (VaR), as well as Conditional Value at Risk (CVaR) have been selected. Using the Standard Deviation for scenario-based portfolio optimization resembles the classical Markowitz approach by calculating

$$\rho = \sigma = \sqrt{\sum_{i \in S} p_i (\ell_i - \mathbb{E}(\ell))^2}. \tag{6.14}$$

The Value-at-Risk at level (1-α) is the α-Quantile of the loss distribution. This risk measure gained significant importance, especially for regulatory purposes. For discrete distribution it equals the sum of the α's smallest values of l_x. While the Mean-VaR optimization problem is non-convex, an evaluation of the VaR value of a distribution is straightforward.

$$\rho = \mathrm{VaR}_\alpha = \inf\{l \in \mathbb{R} : \mathbb{P}(l > \ell) \leq 1 - \alpha\} = \inf\{l \in \mathbb{R} : F_l(\ell) \leq \alpha\} \tag{6.15}$$

The Conditional Value-at-Risk (CVaR), which has been used as a substitute for VaR because a linear programming reformulation is available, and this risk measure additionally exhibits the property of being a coherent risk measure. It is the expectation over the quantile (VaR) of the loss distribution, i.e.

$$\rho = \mathrm{CVaR}_\alpha = \mathbb{E}(\ell | \ell \leq \mathrm{VaR}_\alpha) \tag{6.16}$$

These three risk measures result in three different optimization program formulations, e.g. quadratic optimization in the case of Markowitz, i.e. Standard Deviation. For CVaR, a linear programming reformulation for a finite set of scenarios and a variety of optimization heuristics to solve the Mean-VaR portfolio optimization problem exists. Of course, these reformulations are only valid if standard, convex, non-integer constraints are added, and however different approaches are necessary. However, the scenario-based evolutionary algorithm framework enables a common treatment of risk measures in one coherent way.

6.6.5 Results

A random initial population of size 500 is created by setting the maximum number of zeros in evoGArealInitial to a- 1. Each new population is created by adding the 50 best chromosomes of the previous population, furthermore adding 100 children each of intermediate crossovers as well as those of 1-point crossovers from two random parents drawn out of the best 50 chromosomes. 50 set-zero mutations as well as factor mutations with f = 2 out of the 100 fittest were also added. Finally, 100

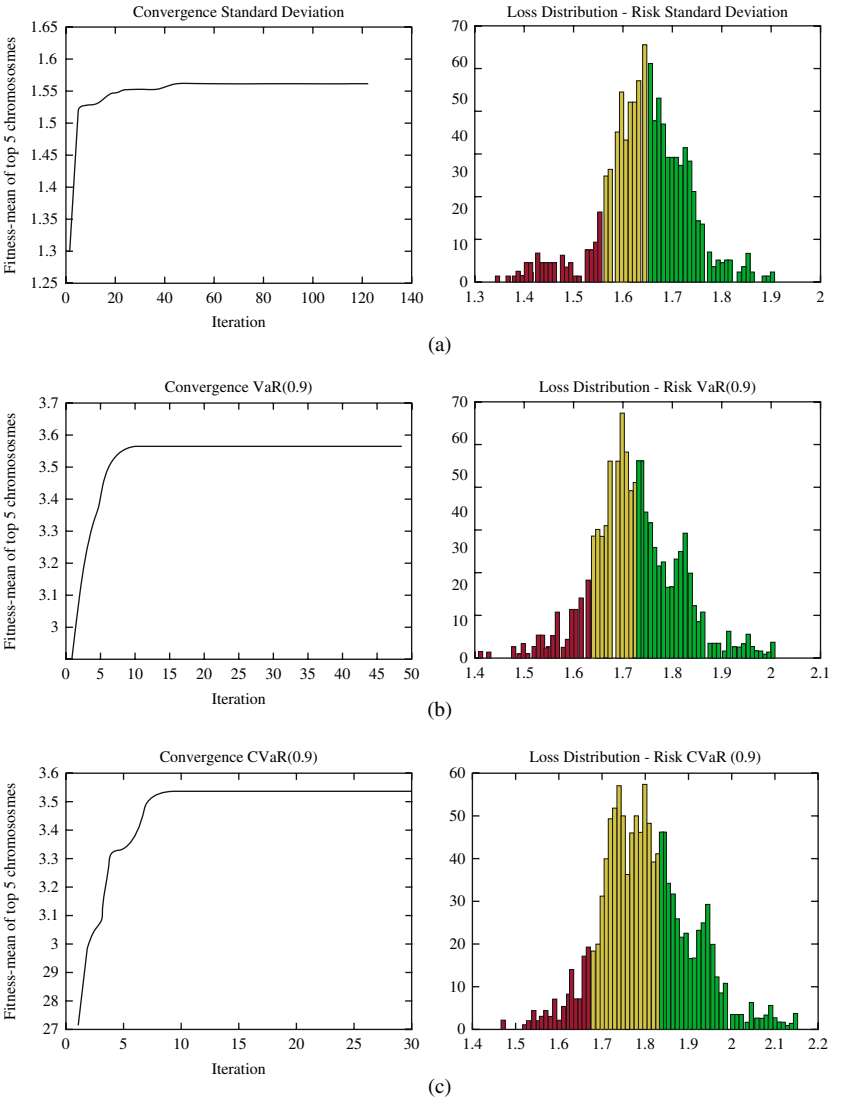


Fig. 6.17 CVaR: Convergence and final loss distribution

2-Point crossovers between one parent randomly drawn out of the 50 best and one out of the 50 worst ones have been added, as well as 100 random chromosomes, again with maximum of a-1 randomly chosen zeros, to complete the offspring population. Each randomly generated or mutated chromosome, and crossover child is delta-normalized with $\delta = 0.05$. The iteration is conducted until the difference of the mean of the fitness value of the 5 fittest chromosomes compared to this value 20 iterations before is smaller than 1e-7.

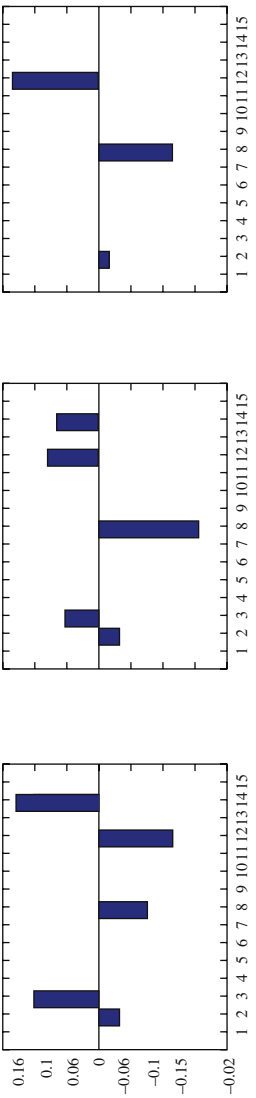


Fig. 6.18 Portfolio comparison: (a) Markowitz/VaR, (b) Markowitz/CVaR, (c) VaR/CVaR

In the case of plain risk-return optimization, the algorithm converges fast and stable, and produces expected results, as shown in Figure 6.17 below. These figures contain convergence and the loss distribution of the final optimal portfolio respective to the chosen risk measure. The quantile α has been set to $\alpha = 0.9$ for both the VaR and the CVaR optimization. In Figure 6.18 a comparison of the computed optimal portfolios is shown, visualizing the differences in asset weights each between two out of the three risk measures.

6.6.6 Conclusion

In this application, an evolutionary computation approach for solving the stochastic scenario-based risk-return portfolio optimization problem has been shown. In contrast to the previous methods mainly based on the multi-criteria aspect of the portfolio selection problem, the Markowitz approach (Correlation), or specialized heuristics for Mean-VaR optimization, this method is plainly based on the loss distribution of a scenario-set and can thus be conveniently applied to a general set of risk measures. One major advantage is the possibility to compare a set of risk measures in one coherent optimization framework, regardless their structure, which would normally necessitate the integration of linear, convex (quadratic), and global optimization or heuristic techniques.

Numerical results have been shown to validate the applicability of this approach. Furthermore, the implementation has been done from the scratch in MatLab and is available under an Academic Free License, such that new extensions can be applied without the need for a reimplementation of the basic portfolio problem.

Future research includes a study of issues, which might occur during the optimization procedure, when special non-convex constraints are added. Furthermore, a comparison between classical convex portfolio optimization approaches as well as other heuristic approaches with the scenario-based evolutionary portfolio optimization techniques presented in this application could prove to be an additional argument to promote the use of nature-inspired algorithms for practical financial problems. While some specific issues of the portfolio optimization problem, i.e. the usually high number of asset-weights being zero, have been included in standard genetic algorithm operators, another valuable extension would be the application of e.g. hybrid mutation schemes, which are based on a preliminary analysis of the scenario set.

Chapter 7
Applications of Genetic Algorithms

Learning Objectives: On completion of this chapter the reader will have knowledge on applications of Genetic Algorithms such as:

- Assembly and disassembly planning by using fuzzy logic & genetic algorithms
- Automatic synthesis of active electronic networks using genetic algorithms
- A genetic algorithm for mixed macro and standard cell placement
- Knowledge acquisition on image processing based on genetic algorithms
- Map segmentation by colour cube genetic k-mean clustering
- Genetic algorithm-based performance analysis of self-excited induction generator
- Feature selection for ANNs using genetic algorithms in condition monitoring
- A genetic algorithm approach to scheduling communications for a class of parallel space-time adaptive processing algorithms
- A multi-objective genetic algorithm for on-chip real-time adaptation of a multicarrier based telecommunications receiver
- A VLSI implementation of an analog neural network suited for genetic algorithms

7.1 Assembly and Disassembly Planning by Using Fuzzy Logic & Genetic Algorithms

The rapid development of new products has shortened product time-to-market and shelf-life, increasing the quantity of wasted used goods. In this context, all these factors must be considered during the design stage of a product. Design for X (DFX) is a collection of methodologies which allow the correct evaluation of several product characteristics and requirements at the earliest stage of the development cycle by enclosing several design aspects (i.e. assembly, disassembly, manufacturability, etc.). Assembly and disassembly are two processes that receive a lot of benefits from DFX. The assembly process is one of the most time-consuming and expensive manufacturing activities. Disassembly aspects have to be taken into

account in several steps of the product Life Cycle, both in product design and during the process design for the disassembly of End-of-Life products. The main objectives of disassembly are the maintenance, remanufacturing, recycling or disposal of end-of-life products. As the complexity of products and production systems increases, the need for computer mediated design tools that aid designers in dealing with assembly and disassembly aspects is becoming greater. The development of efficient algorithms and computer aided integrated methods to evaluate the effectiveness of assembly and disassembly sequences are necessary. Efficiency and flexibility to operate with the maximum number of different products, production environment and plant layouts are the main features of these algorithms. In this application a hybrid Fuzzy Logic–Genetic Algorithm approach is proposed to implement the automatic generation of optimal assembly and disassembly sequences. The aim of this methodology is the efficient generation of these sequences while preserving the flexibility to operate with a great variety of industrial products and assembly/disassembly environments.

7.1.1 Research Background

The realization of an efficient assembly/disassembly (A/D) process is a key factor for the competitiveness of successful company. Environmental regulations and customer pressure to make more environmentally friendly products, force companies to further integrate A/D into the manufacturing environment. However, the efficiency of existing A/D processes is still low because of the inherent difficulty to develop fully automated systems and the complexity of performing operations for a wide range of products. Several methodologies have been proposed by academic and industrial researchers in order to implement automated A/D systems. These methodologies fall into three main research areas related to:

(i) the conception of flexible A/D cells
(ii) design and develop of innovative A/D equipments and tools
(iii) implementation of more efficient control systems
(iv) formalization of predictive models for A/D planning.

The implementation of a fully automated A/D cell requires a medium-high capital investment. The main reason for automating A/D cells is their flexibility to cope with a great number of factors such as: product variety, small batch sizes, different product states, missing components and sometimes inadequate disassembly tools. The cell can also work with new equipment and tools. In this way the disassembly cycle can be shortened using special devices for separating, unscrewing and grasping. In addition specialized active sensors and/or devices must be studied and realized to assure the necessary reliability and speed during on-line product inspection because of the limited time for data acquisition and elaboration. For this reason, a stereoscopic vision system is considered more efficient than others for recognizing product features. Another important aspect is the integration between the several

hardware elements of the disassembly cells (robots, tool changing station, etc.) and disassembly planning software. Such software, normally tailor-made and based on the specific application (e.g. mechanical or electronic parts), require the formalization of an extensive process-oriented knowledge. This knowledge can be embedded into the system through procedures applied to the verification model obtained from the CAD system in order to make comparisons with the real product. The inspection system verifies the state of maintenance of the product. Recognized differences between the product model and physical part are then transferred to the planner and the generation of new optimal sequences is carried out.

Assembly and Disassembly (A/D) Planning

One of the most challenging aspects of A/D is finding the optimal sequence. The assembly sequence is traditionally generated by a human expert who carefully studies the assembly drawing and generates the sequence in his mind. This planning step is very costly and time consuming. Together with time and cost issues, manufacturers are becoming more environmentally sensible. In addition, stricter regulations are forcing manufacturers to become more responsible for the entire product life cycle. As a consequence, the evaluation of disassembly and recycling of products has to be considered at the design stage, planning all the operations to be performed on end-of-life products and taking into consideration different part conditions because of deterioration (i.e. damaged or missing parts).

In the process of disposing and recycling end-of-life products, the cost of handling, sorting and disassembly will play an ever more important role. The aim of assembly planning is the identification of the optimal assembly sequences of products constituted of several parts, whereas disassembly planning is very important during product maintenance or end-of-life applications. In particular, the disassembly sequence necessary to realize efficient product maintenance can be identified by inverting the assembly sequence. This condition is necessary to preserve the product functionality, avoiding destructive operations on components that can lead to useless products. In end-of-life applications, the main aspects influencing the generation of disassembly sequences are the evaluation of the financial and environmental impact of waste (analysis of materials, energy and toxicity is necessary in case of remanufacturing, recycling or disposal) and economic aspects associated with the feasibility of the disassembly process. The development of a system for the automatic recognition of assembly and disassembly features for the generation of sequences is currently in a research phase. In such a system, data on assembly and disassembly features and related resources are retrieved from the CAD and Resource Database in order to plan operation sequences (Figure 7.1). The role of the CAD system is essentially connected to data collection and analysis of the assembly/disassembly process. During data collection, CAD models of the different components are re-organized into one model in order to restore the original configuration of the product. Assembly and Disassembly delations (contact, attachment and blocking between components) and directions are identified. Related resources

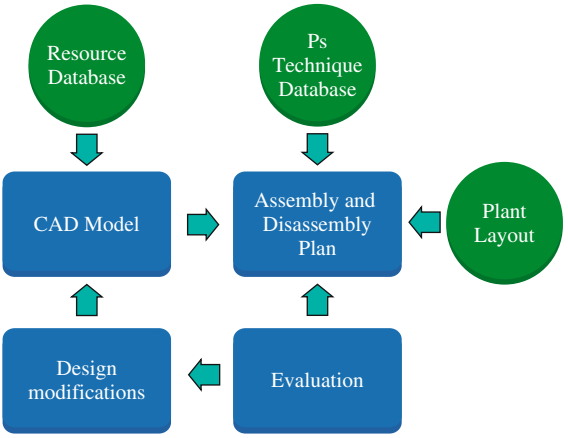


Fig. 7.1 CAD based Assembly/Disassembly system

(grippers and fixtures) linked to disassembly features are then selected to prepare the planning phase. The creation of these sequences, checks on their feasibility and the identification of the optimal sequence are the principal tasks performed by the assembly and disassembly planner. The planner must take into consideration the plant layout for the machine positions. In fact, the machine characteristics and resources influence the shape of the solution of the process (e.g. a change of grippers may have higher costs than re-orientation if performed on one machine, but it may be cheaper if performed on another one). The generated assembly and/or disassembly sequence is then evaluated by designers and process engineers. If an unsatisfactory sequence has been obtained, modifications are performed to the original model and/or to the problem solving technique in order to identify a more effective sequence plan. The successful application of these tasks is strictly dependent on the problem solving techniques used (PS Technique Database). The application of a well developed problem-solving technique allows the reduction of the computational effort without affecting the reliability of the results of the processed model. Moreover, an objective criterion must be used to evaluate disassembly sequences. This function is normally built using some parameters such as the number of product orientations or the number and type of grippers used. Additional constraints and precedence between components can be added to deal with the real industrial environment. Furthermore, easy adaptability to a wide variety of industrial situations is required for the analysis of several products and machines.

Formulation of the Disassembly Problem

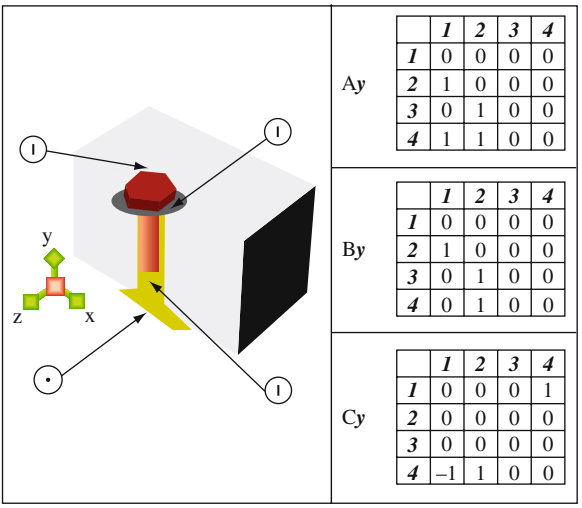
An A/D plan to perform the complete product union/separation is represented using well-specified sets, defined based on product features and manufacturing environment data. These sets generally contain information on the position of each basic component in the A/D sequence (sequence set), tools used to connect/disconnect

interfaces between the selected component and other elements (tool set) and the product orientation chosen during the connection/disconnection tasks (orientation set). The problem of identifying the optimal A/D sequence presents a factorial computational complexity in function of the number of components *n*. This complexity rapidly grows up when the aim is the identification of the optimal A/D plan rather than the sequence. The complexity of A/D planning also becomes dependent on the number of grippers *g* and product orientations *o*. The ranking of the product components in the A/D plan is performed using a properly-designed objective function. This function is built by considering technological aspects, environmental considerations, A/D times and costs, or a combination of them. In addition the proposed solutions must respect constraints related to interactions between components during disassembly, possible grouping of components and directional constraints imposed on the component geometry. In literature, several methods for the problem formulation of A/D planning can be found. This formulation generally consists of two steps. The first one is the definition of the geometrical relationships and mating conditions between components.

The second one is the formalization of the optimization method based on computation, technological and environmental aspects. With regard to the first step, the approaches developed in literature are basically graph based and matrix-based methods. These approaches are equivalent because matrices can be built from graphs and vice versa. In graph-based approaches, relationships between components are represented using the graph semantic; in matrix-based approaches, relations are converted into matrix form. These approaches are very useful if a CAD-based system is to be employed. Main geometry and feature data are extracted from the CAD model and converted into matrix/graph form. A typical application of matrix methods is shown in Figure 7.2, where three matrices (A_y, B_y, C_y) are defined for the considered assembly. The y specifies the considered A/D direction for each matrix. The A_y matrix is called interference matrix, its rows and columns correspond to each component and its elements can be only 0 or 1. Considering the ith row the ones identify the elements that do not permit the disassembly of the ith component in the y direction, the zeros are related to components which do not prevent it. The B_y matrix detects the components in contact with the ith component, while the C_y matrix identifies which components are mechanically connected to the ith component. Then the matrices and the gripper list are then sent to the planner together with the interference matrix. In a computer aided environment this information can be automatically retrieved from the CAD model. Once a component is removed, the row and column associated to it are removed. This process is repeated until only one component is left. The final output is represented by the sequence which optimizes the defined objective function. The second step is the formalization of the optimization method for the generic objective function:

$$f = f(C, T, E)$$

where C, T and E are computation, technological and environmental factors respectively. Computation factors, basically associated to the maximum length of the



Ay	1	2	3	4
1	0	0	0	0
2	1	0	0	0
3	0	1	0	0
4	1	1	0	0

By	1	2	3	4
1	0	0	0	0
2	1	0	0	0
3	0	1	0	0
4	0	1	0	0

Cy	1	2	3	4
1	0	0	0	1
2	0	0	0	0
3	0	0	0	0
4	–1	1	0	0

Fig. 7.2 CAD based Assembly/Disassembly system

feasible sequence, are introduced into the model to perform computations. Technological factors are related to the number of assembly or disassembly operations (gripper changes, product reorientations, fixture changes, etc.). Environmental factors are mainly related to subassembly detection, the theoretical value of the subassembly, weight of the subassembly and percentage of each material present in the subassembly. For a product consisting of a small number of components, the optimal A/D plan can be found using an enumerative algorithm that generates and evaluates all component configurations. This approach becomes unfeasible with a high number of components because of the huge number of combinations. In fact, the factorial complexity grows faster than the polynomial one. Because of the combinational problem complexity, the size of the problem and the flexibility, required to solve the algorithm, influence the choice of the solution approach. An algorithm that works well on small size problems can become impossible to be applied to large problems. On the contrary too complex algorithms can be time and resource consuming. These circumstances lead to several approaches to model and solve the disassembly problem, also to limit the computational effort. The identification of the optimal disassembly plan of a product and/or the creation of viable A/D sequences are commonly performed using off-line methodologies such as:

(i) operational research approaches
(ii) genetic algorithms
(iii) ant colony
(iv) simulated annealing
(v) Petri nets
(vi) a combination of the above

Genetic Algorithms (GAs) are particularly efficient in searching for optimal solutions. GAs are more robust than existing direct search methods (hill climbing, simulated annealing, etc) because they present a multi-directional search in the solution space (set of individuals) and encourage information exchange between these directions. GAs unfortunately show some drawbacks in this application, such as difficulties in finding the optimal value of genetic parameters, problems related to the search of optimal values of the weighted fitness function and difficulties related to multi-objective problems. In order to improve the GA behavior to avoid these drawbacks, an integrated approach with Fuzzy Logic has been proposed.

7.1.2 Proposed Approach and Case Studies

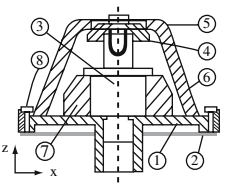
Actually there is an increasing interest in the integration of Fuzzy Logic (FL) and Genetic Algorithms (Gas). According to an exhaustive literature analysis, this integration can be realized on two levels:

(i) FL can be used to improve GA behavior and modeling GA components, resulting in fuzzy genetic algorithms (FGAs);
(ii) the application of GAs in various optimization and search problems involving fuzzy systems (Genetic Fuzzy Systems – GFS).

A Fuzzy Genetic Algorithm (FGA) is a GA in which some algorithm components are implemented using fuzzy logic based tools, such as: fuzzy operators and fuzzy connectives for designing genetic operators with different properties, fuzzy logic control systems for controlling the GA parameters according to some performance measures, fuzzy stop criteria, etc. FGAs are a particular kind of adaptive genetic algorithms (AGAs), characterized by a variation of their features during running based on some specific performance measures. A generic AGA can perform the adaptive setting of parameter values, the adaptive selection of genetic operators and produce an adaptive representation of the fitness function. Among these ways of making a GA adaptive, the adaptation of parameter setting has been widely studied. In particular, rules for discrete Fuzzy Logic Controllers of mutation probability and selection pressure have been formalized (cross-over probability, cross-over rate etc.). According to these rules, a high mutation probability and a low selection pressure must be used during the initial generation of individuals. The second integration method involves techniques such as Fuzzy clustering, Fuzzy optimization (GAs are used for solving different fuzzy optimization problems), Fuzzy neural networks (GAs are used for determining an optimal set of link weight and for participating in hybrid learning algorithms) and Fuzzy expert systems (GAs can solve basic problems of the knowledge base). In this application a hybrid Fuzzy-GA methodology is used to identify the optimal assembly and disassembly sequences while improving performances of GAs. A case study has been selected in order to investigate and validate the proposed approach. The identification of the optimal disassembly sequences of these products was carried using different methodologies. Figure 7.3 reports product configurations, the component list and available grippers to perform A/D operations.

Fig. 7.3 First product

Part ID	Available Grippers
1	G1 G2 G4
2	G1 G2 G4
3	G1 G2 G3
4	G3
5	G1 G2 G4
6	G1 G2 G4
7	G1 G2 G4
8	G3



Some assumption were made to generate optimal assembly and disassembly sequences such as the product must be entirely disassembled (i.e. disassembly can be performed by reversing assembly sequence), the relationships between the components were known before the search of the optimal planning sequence, only technological and computation factors were considered in the fitness function below. The proposed approach was developed starting from a well-known matrix-based GA methodology found in the literature. Here, a generic chromosome consisted of three sections that specify the component identifier, disassembly direction and selected gripper to be used for each part. The evaluation of the disassembly sequences was realized using the fitness function:

$$f = w_1{}^*1 + w_2{}^*(N-1-o) + w_3{}^*(N-g-1)$$

where constant N is the component number, while parameters *l, o, g, s* are, respectively, the maximum length of the feasible sequence, the orientation change number, gripper change number and maximum number of similar grouped disassembly components. The weights w_i with i =1,3 are associated to each parameter. Moreover an enhanced version with an adaptive fitness function found in was also considered. The difference with the previous fitness function was the creation of a function with a single term directly associated to the maximum length of the feasible sequence. In this way, individuals with a length less than the feasible one are removed from the initial population during evolution. Once the population contains all feasible individuals, the fitness function returns to the form

$$\mathrm{f} = \mathrm{f}(\mathrm{C},\mathrm{T},\mathrm{E})$$

The advantage of using these twin functions was an improvement of GA convergence. The Fuzzy-GA hybridization was implemented by the authors onto two levels. The first level consisted of substituting the algebraic fitness function with a Fuzzy function to obtain closer control of the technological knowledge of the disassembly process. In fact normally there is not a unique definition of the weight affecting each technological factor (re-orientation, gripper changes, etc) and often these weights have to be changed depending on the plant layout and the availability of machines. Fuzzy classification systems based on fuzzy logic are capable of dealing with uncertainties and can be intuitively varied by an operator according to the machine availability. For each parameter, a fuzzy set with 3 triangular membership functions was created to specify the process condition (bad, medium or good). The fuzzy sets of each parameter were then used as input of the Mamdani Inference

system, obtaining a single defuzzified output, used to evaluate the quality of chromosomes generated by the GA. As a consequence, the first advantage of this approach was the use of a non-weighted fitness function, avoiding solution dependency on the disassembly product characteristics because of the weight balancing. Moreover a better response was achieved thanks to the natural attitude of a Fuzzy Logic system to cope with multi-objective problems. The second level of integration considered the fuzzy control of the genetic operators. As mentioned before, the algorithm was dramatically influenced by the imposed mutation probability and crossover rate. The developed fuzzy controller was able to modify these probabilities during algorithm execution. The Fuzzy Logic was embedded into the controller of a smart algorithm. The main features of this controller were the possibility of a better allocation of the algorithm resources to improve its performance, direct on-line tuning of Genetic Algorithm parameters and the comparison between off-line and on-line parameters for better control. The main advantages offered by adapting GA parameters at run-time were the improved search in the solution space and a fast growth of the best chromosome fitness value independently of genetic parameters. A GA based software program was designed and developed in the MATLAB environment. The structure of the algorithm is represented in Figure 7.4. Four choices were available:

A) Pure genetic algorithm with a single algebraic fitness function;
B) Genetic algorithm and a fuzzy ranking function;
C) Pure genetic algorithm with adaptive algebraic fitness function;
D) Adaptive fitness function and dynamic fuzzy controlled genetic algorithm.

7.1.3 Discussion of Results

The experiment was conducted neglecting the influence of the population size on the performance of the GAs. Population size normally has an influence on algorithm

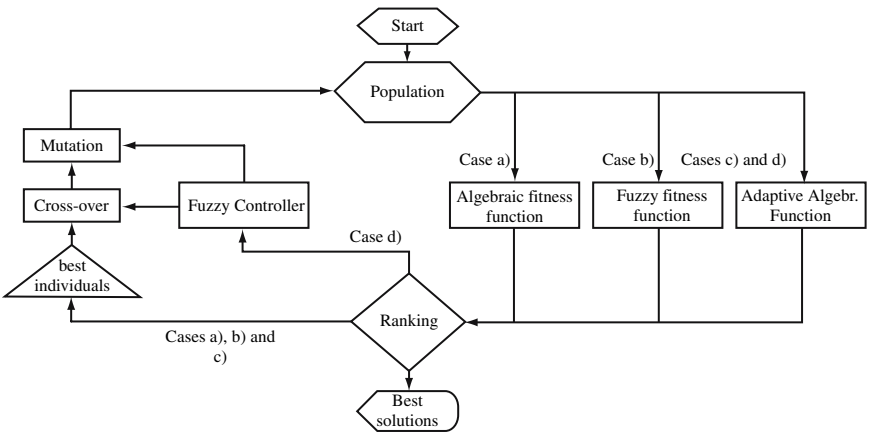


Fig. 7.4 Data flow diagram

performances, but in this study the authors preferred to focus their attention on the aspects connected to the solution sensitivity due to mutation probability and crossover rate variations.

As a consequence the initial population was set to 80 for all runs.

Choice A) versus Choice B)

The pure genetic algorithm with the single algebraic fitness function (2) was compared to the genetic algorithm with a fuzzy ranking function (first level of hybridization). The comparison was carried out by evaluating results obtained from running both GA choices while keeping the same crossover rate, mutation probability and population size. In this way the same conditions were maintained for both investigated case studies. The parameters that led to best GA performance were used during the program running (mutation probability equal to 80% and a crossover rate equal to 40%). The runs were repeated 20 times each and the fitness function used to compare results was the mean of the maximum fitness for all the runs. The maximum fitness versus the generation for the case study is showed in Figure 7.5. Both the curves have similar behavior but the GA seems to behave slightly better than the Fuzzy. At first glance, the performance of the first level of integration seems to promote the use of pure GA. However these results must be carefully analyzed. In fact, a generic Fuzzy Function was employed while the parameters of the GA algebraic function were optimized for the assembly/disassembly applications.

The optimal tuning of GA parameters is not a simple operation but a very time-consuming task because results are only obtained after several GA runs have been performed. This task is applicable for products with a low number of

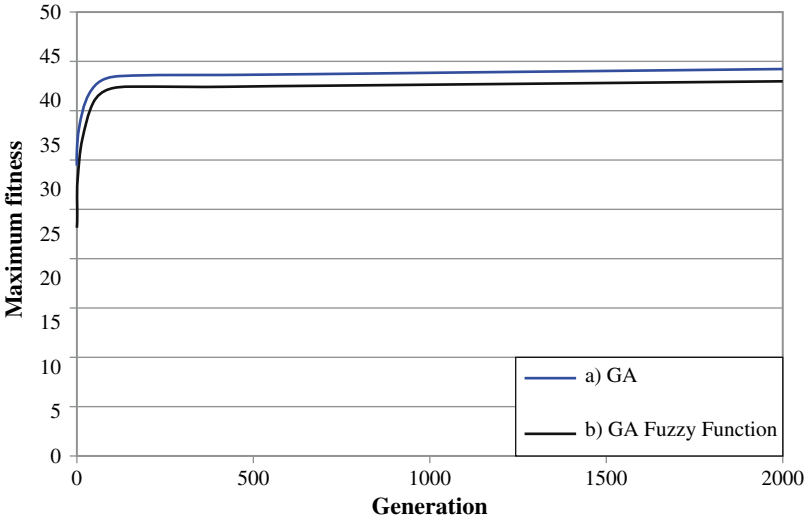


Fig. 7.5 Max. fitness vs generations

components but becomes unfeasible, increasing of the component number. Another aspect to consider is related to the number of successful identifications of the optimal sequence. Considering the second case study, the GA identified the optimal A/D sequence on 25% of the runs while GA with the Fuzzy function succeeded for 60% of the runs.

The lower value of GA with Fuzzy function at the last generation can be explained by the fact that unfeasible sequences were obtained. The fitness of these sequences lowered the mean fitness function value used during comparison. On the contrary, the shape of the fitness function of GA eliminated unfeasible solutions and led to optimal or near-optimal solutions.

Choice C) versus choice D)

The performance of the Fuzzy controller was compared with the performances of GA with the adaptive fitness function. The comparison between these algorithms was performed by running algorithm C) 16 times (crossover rate 0.2–0.8 with step 0.2 and mutation probability 0.2–0.8 with step 0.2) for both case studies. The population size of the second case appears to be under-sized with respect to the first case. The result of the case study is shown in Figure 7.6. Different behaviour of the GA appears in both the situations.

During the identification of the optimal sequence of the first assembly the GA seems to behave well with a mutation probability equal to 0.8, independently of the value of the crossover rate. However, the result obtained using the Fuzzy-controlled

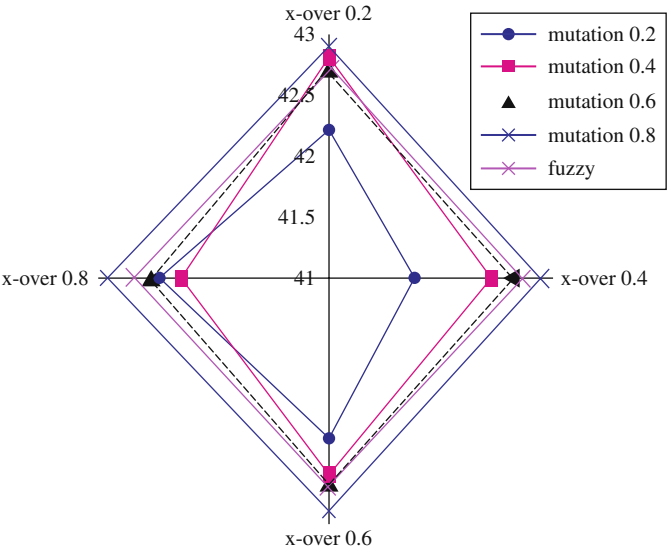


Fig. 7.6 Cross-over rate and mutation

Table 7.1 Assembly sequence

$8 \rightarrow 7 \rightarrow 5 \rightarrow 6 \rightarrow 4 \rightarrow 2 \rightarrow 3 \rightarrow 1$
$+z \rightarrow +z \rightarrow +z \rightarrow +z \rightarrow +z \rightarrow +z \rightarrow +z \rightarrow +z$
$G3 \rightarrow Gl \rightarrow Gl \rightarrow Gl \rightarrow G3 \rightarrow Gl \rightarrow Gl \rightarrow Gl$

GA was better than those of pure GA and near to hose obtained using the GA best performances in most cases. In Table 7.1 the optimal sequences found are shown for the assembly.

7.1.4 Concluding Remarks

Considering the results obtained using Fuzzy-GA, the stability and flexibility of the Fuzzy-GA approach for product assembly and disassembly planning can be further enhanced by studying a wider variety of product types and configurations. In particular the first level of hybridization gave similar results to the GA with the best parameter settings. On the other hand the second level denoted satisfying results, which can be enhanced with several improvements. The Fuzzy function can be improved by making it adaptive and more problem oriented. The controller can also be improvable by making the population size and crossover probability adaptative as well.

7.2 Automatic Synthesis of Active Electronic Networks Using Genetic Algorithms

Numerical optimization is rightly considered to be a valuable tool for synthesizing electronic networks. By minimizing a suitable objective function it is possible to determine component values which provide the best performance measured against some target specification. Unfortunately numerical optimisation suffers from the serious limitation that it only operates on networks of fixed topology. Selecting a suitable network configuration is usually the most difficult part of electronic network design, and this must be completed manually before numerical optimisation can take place. Genetic algorithms (GAs) are a class of optimisation methods which have been successfully applied to a wide range of numerical and non-numerical optimisation problems. Based on the principle of "survival of the fittest", GAs have proved to be both robust and efficient. Providing that a method exists for analysing design solutions, and for measuring their performance against predetermined criteria, GAs can be used to close the design feedback loop. Methods for analysing electronic networks in both the frequency domain and time domain are well established. It is to be expected, therefore, that GAs should be suitable for optimising the topology of electronic networks, and this has been found to be the case.

Passive networks containing resistors and capacitors can only generate voltage transfer functions with real poles, and this seriously restricts the frequency-domain and time-domain behaviour. Including inductors in the network removes this restriction, but inductors suffer from a number of drawbacks (including cost, size and non-linearity), particularly at low frequencies. Fortunately RC-active networks do not suffer from this limitation and can be used to realise any stable voltage transfer function. Below about 100 kHz RC-active networks are preferable to LCR networks. This application describes the use of GAs to synthesise active electronic networks containing a single operational amplifier.

7.2.1 Active Network Synthesis Using GAs

Operational amplifiers have long been established as the preferred active element in low-power linear networks at frequencies below about 100 kHz. There are many reasons for this, including their near-ideal performance and low cost. It was therefore decided to use operational amplifiers as the active elements in automatic network synthesis. The network synthesis in fact uses infinite-gain four-terminal operational amplifiers, or nullors (nullator/norator combinations) which have two input terminals and two output terminals. The reason for using nullors is that they simplify network analysis. Real operational amplifiers have only a single output terminal, and can be modelled by a nullor with one output terminal grounded. Although the eventual goal is to synthesise networks containing several operational amplifiers, this is a far more complex task than synthesising passive networks. A decision was therefore taken to limit the scope initially to networks containing a single operational amplifier. The synthesis starts with a skeleton network in which the network input node, network output node, network ground node and nullor nodes are fixed. The nullor inputs are distinct from the other fixed nodes; one nullor output is the ground node (so that the synthesised networks can be constructed from real operational amplifiers) and the other nullor output node is the network output node (to provide a low output impedance). Passive elements are then added to the skeleton network under control of the GA. In traditional GAs the chromosome is a binary string which is decoded to obtain the design solution, and much of the original theoretical work on GAs assumed binary coding. However, there is growing evidence that other forms of coding may be equally effective. Certainly the use of a binary-coded chromosome in network synthesis leads to the generation of an unacceptable proportion of non-viable networks. Consequently a representation is used in which the chromosome consists of a number of genes, each of which corresponds to a passive component in the network. The GA therefore operates only on the passive elements of the network. The method of coding used is to associate each gene with one passive component (resistor, capacitor or empty). A complete chromosome consisting of a number of genes defines the passive part of the network. Taken in conjunction with the skeleton network, this constitutes a complete active network which can be analyzed and its response compared with the target specification. The GA selects

the network topology only, component values being determined later by numerical optimisation. Consequently the genes need specify only the component type, and a pair of terminal nodes. Chromosomes consist of a fixed number of genes, whilst networks contain a variable number of components. This discrepancy is overcome by the inclusion of an "empty" component type:

```
typedef enum { resistor, capacitor, empty } comp_type;
typedef struct {
node n1, n2;
comp_type typ;
} gene;
typedef gene chromosome[chrom_size];
```

Each individual of the initial population is a connected network consisting of random component types (excluding the "empty" type) with random terminal nodes. New members of the population are generated either by breeding from a pair of individuals, or by mutation of a single individual. The proportion of new individuals generated by breeding is determined by the parameter **xover_prob**. Fitness ranking by the tournament method, is used to select parents. A number of potential parents (normally 3) are selected randomly from the population. The fitnesses of these parents are then compared and the successful individual is the one with the highest fitness. This process is repeated for the other parent. Chromosomes are combined by uniform cross-over: each gene in the offspring's chromosome is derived from one of the two parents, selected at random with equal probability. A proportion of the offspring is then subjected to mutation as described below. The complete mutation process may involve several basic mutation operations:

```
mutate_chrom(offspring);
while (flip(mutate_prob))
mutate_chrom(offspring);
```

where **flip(x)** returns a value **true** with probability **x**. In order to reduce the number of lethal mutations a procedure has been adopted which normally results in a viable, although not necessarily fit, individual. Each call to **mutate_chrom()** applies one of the following network transformations, selected randomly, but with equal probability:

1. Replace an existing component by an open circuit.
2. Replace an existing component by a short circuit.
3. Add a random component type in parallel with an existing component.
4. Add a random component type in series with an existing component.

By repeated application of these four transformations any network configuration is accessible from any other configuration. Since these basic mutations may be applied successively it is theoretically possible (although extremely unlikely) for any network to mutate to the optimum network in a single complete mutation operation.

Cross-over and mutation are complicated in the case of active networks because they can lead to pathological networks. For example, replacing a passive component by a short-circuit could lead to the input or output terminals of the nullor becoming short-circuited. Following cross-over and mutation the network is checked for these and other conditions; if necessary the cross-over or mutation process is repeated until a viable network is generated. Offspring are checked against the existing population to determine whether they are identical to any existing individuals. Duplicates are rejected. The first stage in fitness evaluation is to decode the chromosome into a form suitable for network analysis. Passive components from the genes are combined with the skeleton network containing the nullor, and the result stored in the form of network graphs. Then the component values which give the best fit to the target response are determined by numerical optimisation. Random starting values are used for the optimisation. The response of the optimised system is compared with the target at a number of points, the sum of squares of the errors is calculated and the fitness taken to be the reciprocal this sum. Left unchecked the GA will tend to generate successively more complex networks until they exceed the analysis power of the computer, because in general a complex network will provide a better fit to any target response than a simple one. To offset this tendency the fitness calculated as described above is multiplied by a penalty function which is close to unity for simple networks, but which rapidly becomes smaller for networks whose complexity exceeds some predetermined level. A typical penalty function p might be:

$$p = \frac{1}{1 + e^{n-8}}$$

where n is the number of passive components in the network.

Rather than create a complete new generation, a steady-state replacement strategy is used: the offspring replaces the least-fit member of the existing population provided that the fitness of the offspring is superior.

7.2.2 Example of an Automatically-Synthesized Network

The example presented here is a network synthesised to meet a frequency-domain specification. The target response is:

$$\left|H_g(j\omega)\right| = \sqrt{\frac{2\pi}{\omega}}$$

over the frequency range:

$$0.1 \leq \frac{\omega}{2\pi} \leq 10.0$$

This response cannot be synthesised by a passive (transformerless) network because it has a voltage gain of greater than unity over an extended frequency range.

A population of 20 individuals was used, and the GA was run for 400 generations. This took around 1 hour on a 100 MHz Pentium PC. On completion several of the fittest individuals had nearly equal fitness values. Further investigation revealed that these different networks produced similar voltage transfer functions. The synthesised network is shown in Figure 7.7. Component values are normalised and impedances would, of course, be scaled appropriately before implementation. It is difficult to see how this network could have been arrived at by any traditional design process. Figure 7.8 shows the frequency response of the network, together with the target response. At no point within the specified frequency range does the actual response differ from the target response by more that 0.1 dB. In practice, of course, component tolerances would probably prevent such an accurate response being obtained from a real circuit. Nevertheless, the network does not seem to have an unduly high sensitivity to component values. If low sensitivity were a requirement, then the network sensitivity could be evaluated and incorporated into the fitness function.

It is clear that this result could have been obtained by mutation alone, without the need for crossover, because the basic mutations may be applied successively, as already described. However, the chances of a single complete mutation operation leading directly to the optimum network configuration is vanishingly small. It is difficult to predict what effect crossover will have on the overall efficiency, and so a large number of syntheses were performed with different values of **mutate_prob** and **xover_prob**. The results broadly indicate that even a small amount of crossover improves the efficiency, but that no consistent improvement is to be obtained by using values of **xover_prob** greater than about 0.3. The values of **mutate_prob** were also found to be non-critical, with the best results being obtained between 0.4 and 0.9. Active networks have also been synthesised to meet time-domain specifications. Calculating the time-domain response of a network is more complicated than calculating the frequency-domain response, and synthesis therefore takes longer. Typically the synthesis of a network containing 8 passive components and one operational amplifier takes 3 hours.

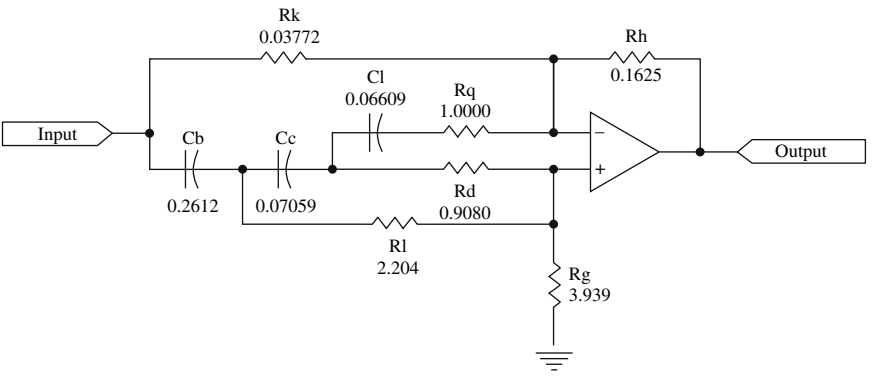


Fig. 7.7 Filter generated for frequency-domain specification

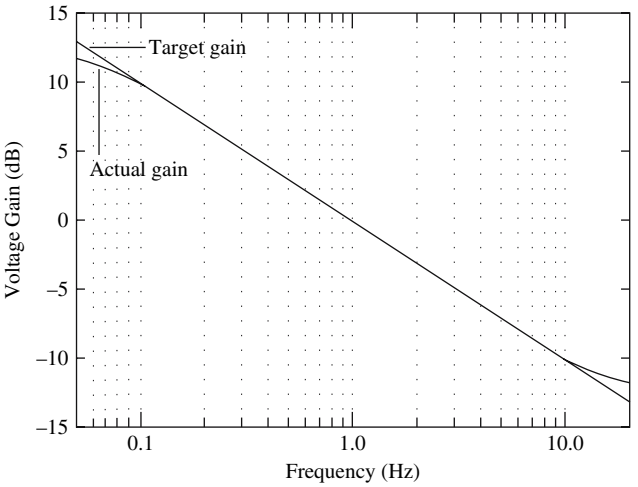


Fig. 7.8 Network generated for the frequency-domain specification

7.2.3 Limitations of Automatic Network Synthesis

There is no reason in principle why GAs should not be applied to the synthesis of any type of network. If a network can be analyzed, then it can be optimised. In practice, however, the amount of computation required limits the type of synthesis that can be performed. In particular the analysis of non-linear networks involves numerical solution of non-linear differential equations, a very computationally expensive process, and synthesis of non-linear networks is probably impracticable. Even for linear networks the amount of computational effort increases rapidly with network complexity. Using a PC, and compiling to 32-bit Pentium code, it is possible to synthesise networks with up to 14 nodes in around 8 hours. With the ever more powerful PCs becoming available the size of networks that can be synthesised is likely to increase as time goes by. Although this application has been concerned with synthesis of networks containing an operational amplifier, other types of controlled source (VCVS, VCCS, CCVS, CCCS, CCI, CCII) can be modelled by combinations of nullors and passive components. Minor modifications to the synthesis method for operational amplifiers should allow synthesis of networks containing these alternative active elements. It should also be relatively straightforward to extend the technique to deal with networks containing several operational amplifiers.

7.2.4 Concluding Remarks

Genetic algorithms provide an effective technique for automatic synthesis of active analogue networks containing a single operational amplifier. The synthesised

networks are highly effective and would not result from any established design procedure. It should be possible to extend the technique to networks containing several operational amplifiers, and to networks containing other types of controlled source.

7.3 A Genetic Algorithm for Mixed Macro and Standard Cell Placement

An important part of VLSI circuit design is the placement process. Many contemporary systems use a combination of macro cells and standard cells in order to implement all the desired functions on a chip: this is the *mixed macro and standard cell* design style. In a typical design, there are usually many more standard cells than macro cells. Thus, before the placement procedure is performed, the standard cells are partitioned into groups, or *domains*, to reduce the total number of components to be initially placed. Contemporary mixed macro and standard cell placement tools use a top-down, multiple-stage approach to determine the arrangement of components on a chip. During the first stage, *block placement*, the arrangement of the macro cells and domains is determined. The dimensions of each domain have not yet been defined, so they must be estimated before block placement can be performed. After the block placement stage has been completed, a *cell placement* stage is applied to each domain to determine the arrangement of its standard cells. Unfortunately, there are often significant differences between the estimated domain dimensions and the actual dimensions after cell placement. When domain estimation errors occur, it becomes necessary to adjust the domain dimensions and repeat the placement process. For a large circuit with many domains, there may be many repetitions of this process before the estimation errors are within acceptable levels of tolerance. One way to address this limitation is to incorporate the relationship between the block and cell levels into the placement process. This application describes the development of a *genetic algorithm* placement tool that uses this relationship to determine an optimal mixed macro and standard cell layout.

7.3.1 Genetic Algorithm for Placement

Genetic algorithms are heuristic optimization methods whose mechanisms are analogous to biological evolution. A genetic algorithm starts with a set of *genotypes*, which represent possible solutions for a given problem. Each genotype is assigned a *fitness value*, based on how well the genotype meets the problem objectives. Using these fitness values, genotypes are randomly selected to be *parents*. These parents form new genotypes, or *offspring*, via *reproduction*. Parent selection and reproduction operations continue for several iterations until the algorithm converges to an

optimal solution. For the placement problem, previous applications of genetic algorithms include Esbensen's method for macro cell placement and GASP for standard cell placement. Each method uses a particular genotype structure to represent the layout for a given design style. A GAP (Genetic Algorithm for Placement) is developed for mixed macro and standard cell netlists. This application builds upon previous efforts by expanding the genotype structures to handle both macro cell and standard cell layout. First, GAP uses the hMetis program to partition a netlist into a specified number of domains. Then, GAP uses a genetic algorithm to produce a valid mixed macro and standard cell layout. Figure 7.9 shows the structure of this genetic algorithm.

Genotype Structure

The partitioning process forms a *cell membership tree*: each leaf corresponds to a cell in the netlist, while the internal nodes represent domains. GAP constructs a *genotype template* by proceeding top-down on the cell membership tree. Each level of the tree has its own encoding to represent a valid layout for the particular level. An example of a GAP genotype is shown in Figure 7.10, while Figure 7.11 shows its corresponding layout. This genotype is for a small netlist with two macro cells (A and B), and ten standard cells (1–10) that are grouped into one domain (C). Thus, the genotype template tree has a system node (block level) with one domain node C (cell level). First, an encoding is randomly generated for the system node. For the *block level* layout representation, GAP uses Esbensen's genotype structure: a *Polish expression* is randomly generated to represent the block *topology* (relative locations), while a *bitstring* is randomly generated to represent the block *orientations*. In Figure 7.10, the Polish expression is $AB+C^*$, while the bitstring is 001 000 111. This means that macro cell A is below macro cell B, while domain C is to the right of A and B. Macro cell A has orientation 1 (rotated 90 degrees), macro cell B has orientation 0 (no rotation), and domain C has orientation 7 (rotated 270 degrees and reflected across the y-axis). The *cell level* consists of standard cells within a

```
construct population of 200 genotypes
evaluate population
for g = 1 to 100 do
        for i = 1 to 50 do
                select parents
                create offspring
                with probability P{0.015} do
                        mutate offspring
                add offspring to population
        end for
        evaluate offspring
        eliminate the 50 lowest-ranked
                genotype from population
end for
final solution = top-ranked genotype
```

Fig. 7.9 GAP genetic algorithm

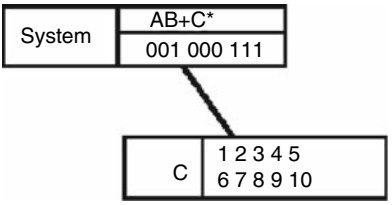


Fig. 7.10 Genotype encoding example

domain. GAP uses the genotype structure of GASP: an *ordered string* is randomly generated to represent the standard cell topology within the domain. In Figure 7.11, the ordered string for domain *C* is $\{1\ 2\ 3\ 4\ 5 \mid 6\ 7\ 8\ 9\ 10\}$. This means that domain *C* has two rows of standard cells: row 1 contains cells 1–5, while row 2 contains cells 6–10.

Fitness Evaluation and Parent Selection

Each genotype is evaluated according to a *fitness function*. This function determines the quality of the solutions that are represented by the genotypes, with respect to the problem objectives. For mixed macro and standard cell layout, the objectives are to minimize the layout area and interconnection wire lengths. Thus, each genotype must be decoded to yield an area cost and a wire length cost for the fitness function. The *area cost* is computed as the product of the layout height and width, while the *wire length cost* is computed as the sum of all wire lengths for the nets in the layout. Since nets will not be routed until after the placement process has been completed, the wire lengths are estimated using the half-perimeter wire length metric.

GAP uses the approach of Esbensen for fitness evaluation and parent selection. Genotypes are ranked based on Pareto dominance using the method of; Whitley's rank-based selection method is then used for parent selection based on these dominance-assigned ranks.

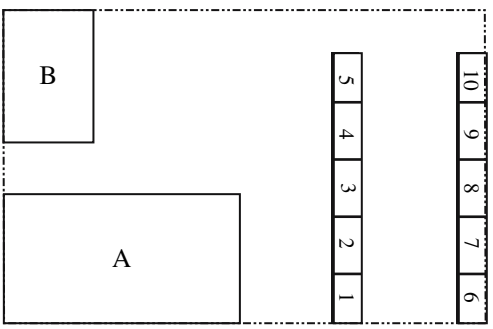


Fig. 7.11 Layout represented by genotype example

Reproduction

Reproduction consists of crossover and mutation. The crossover operator creates a new genotype, based on the structures of two parent genotypes. Since each parent genotype tree follows the structure of the genotype template, the offspring genotype tree must also follow this structure. The main difference between the offspring and its parents will be the encoding for each node on the genotype tree. Figure 7.12 shows two parent genotypes (a) *P1* and (b) *P2*, and the resultant offspring (c) *Q*. The root of each genotype tree is the system node, and the node encoding is a combination of a Polish expression and a bitstring. For offspring Q, the corresponding root node is created by applying Cohoon's crossover operator to the Polish expression and uniform crossover to the bitstring. The next node on the genotype trees is domain C; this has an ordered string encoding. Therefore, the corresponding node on the offspring genotype *Q* is created by applying cycle crossover. An additional operation is to determine the number of rows *r* for the offspring *Q*. The number *r* is randomly generated: $1 \leq r \leq$ #cells. For this example, $r = 5$.

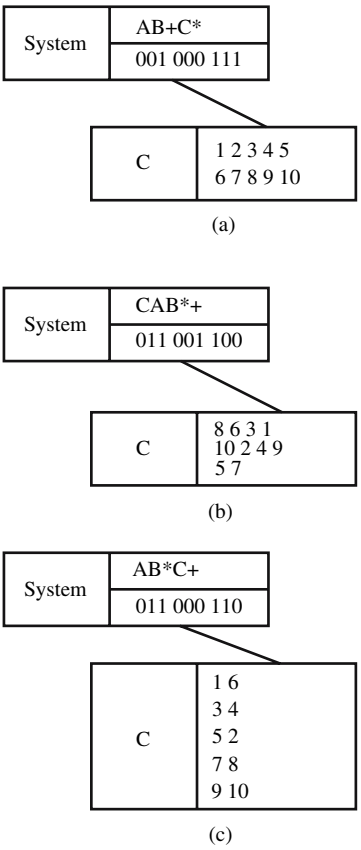


Fig. 7.12 Crossover on (a) and (b) to form (c)

An offspring may also undergo *mutation*, which alters its structure. The mutation operator developed for GAP follows the same general principle as the crossover operator: traverse the offspring tree in a top-down manner and apply the appropriate mutation operation to each node encoding. For the system node, Wong's mutation operator is applied to the Polish expression and bitwise mutation is applied to the bitstring. At each domain node, pairwise mutation is applied to the ordered string.

7.3.2 Experimental Results

GAP was compared against the previous top-down, multiple-stage placement approach. In order to provide a fair comparison, Esbensen's placement tool was used for the block placement stage of the top-down approach, while GASP was used for the cell placement stage. The methods were tested on the mixed macro and standard cell benchmark netlists *a3*, *g2*, and *t1*. Each netlist was partitioned into ten domains, and ten trials were run for each method on each data set. Figure 7.13 shows the mean area response for each netlist, while Figure 7.14 shows the mean wire length response. Compared to the top-down, multiple-stage approach, GAP yielded an average of 27% improvement in layout area and an average of 10% improvement in layout wire length. The reason for these improvements is most likely due to the elimination of the need for domain size estimation in GAP. Domain size estimation methods tend to overestimate dimensions in order to allow space to complete standard cell placement within the domain. Large domain dimensions result in large layout areas and increased interconnection wire lengths. In contrast, GAP is able to obtain the actual domain size since this information is encoded in its genotype structure. This eliminates the overestimation of domain sizes, and results in reduced layout areas and wire lengths.

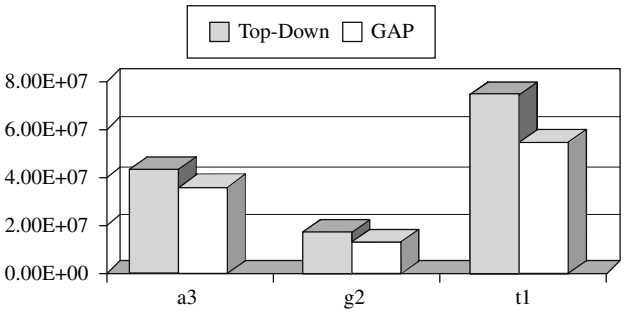


Fig. 7.13 Mean area response

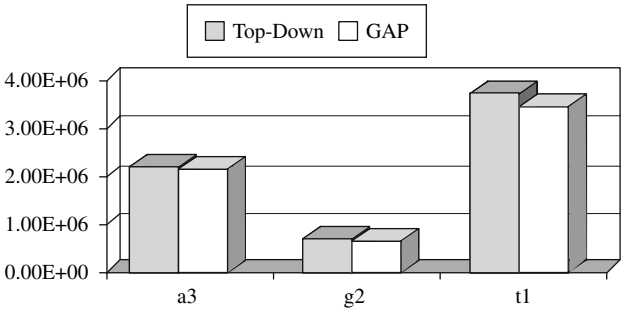


Fig. 7.14 Mean wire length response

7.4 Knowledge Acquisition on Image Procssing Based on Genetic Algorithms

Easy and immediate acquisition of large numbers of digital color images, for example, of the daily growth of plants in remote fields, has been made possible via the Internet nowadays. From such images, detailed information concerning the shape, growth rate and leaf colors of plants are obtained. Vast quantities of image data, however, increase the time spent extracting such information from the data. This is because the extraction procedure needs human aid – empirical knowledge of image processing and the features of target objects. Thus, image analysis, segmenting images of objects and deriving their outlines or areas, commonly invokes procedures based not only on routine, but also on trial and error performed by hand. Automated image processing systems, such as expert systems, have been studied in various areas of engineering. Nazif and Levine, Mckeown have studied rule-based expert systems for an automated segmentation of monochrome images, linked to a knowledge base on target components in the images. Those systems, however, have had common difficulty that they must implement vast networks of complex rules, and exercise empirical knowledge on image processing and the features of targets. To overcome such difficulty, automated and easy optimization of procedures for image segmentation has been studied. Fortunately, color image data can be easily obtained now. Since color images have considerable information for segmenting targets, the rules for automated segmentation can be simplified. In this application, the procedures for selecting filtering algorithms and for adjusting their parameters to segment target components in images is emphasized. Genetic algorithms (GAs) are suitable for this purpose because the algorithms involve optimization and automation by trial and error.

7.4.1 Methods

The methods are discussed as follows:

Image Segmentation Strategy

Many kinds of efficient filtering algorithms for image segmentation, such as noise elimination and edge enhancement, have been contrived. Implementing all of them into the algorithms, however, is unrealistic because the increase in operations invokes a proportional increase in processing time. Based on the empirical knowledge of the segmentation of plant images, several filtering algorithms that are commonly used have been selected, and implemented in the algorithm as shown in Table 7.2. The thresholding and reversion algorithms are performed on a focused pixel of the image processed in serial order, and others have spatial mask operators.

While procedures for edge enhancement of monochrome images are available, common procedures to segment targets in color images are considered. The procedures in Figure 7.15 are explained as follows:

1) Color of component areas in the images is averaged using smoothing (SM).
2) Target components are enhanced using thresholding on hue (TH) and, simultaneously, the image is entirely converted to a monochrome image.
3) Differentiation (EE) is used when target features outline components.
4) Binarization (TB) is performed for the entire monochrome image.
5) Reversion (RV) on binarized pixels is occasionally effective to enhance the components.
6) Fusion operations, expansion (EF) and contraction (CF), allow a reduction in noise, and occasionally, is performed repeatedly.

After these procedure are carried out, the image processed has been converted to a binarized image with target components defined. In the algorithm, the HSI model is adopted, because the it is efficient for the segmentation of plants in fields. All operations are performed after each pixel value is converted from RGB to HSI. The

Table 7.2 Filtering algorithms used as phenotypes

Manipulation	Algorithms	Symbols
Thresholding (hue)	Point processing	TH
Thresholding (brightness)	Point processing	TB
Smoothing	Median operator	SM
Edge enhancement	Sobel operator	EE
Contraction	4-neighbor fusion	CF
Expansion	4-neighbor fusion	EF
Reversion	Point processing	RV

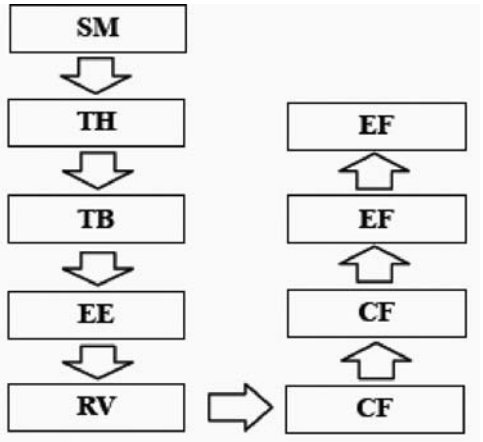


Fig. 7.15 Flow chart of a typical procedure for enhancing targets components

smoothing algorithm is a median operator with a 3*3 mask of pixels, and it is applied only for the hue of the pixels. The thresholding has two different operators; one operates upon the hue and another upon the brightness of pixels. These operations substitute null for all bits of a pixel when the pixel value occurs within a range defined by minimum and maximum values. When the value is out of the range, they substitute unity for all bits of the pixel. For edge enhancement of components in images, a Sobel operator with a 3*3 mask of pixels is used. The operator applied to the brightness value substitutes null for the saturation of pixels to convert the images to monochrome ones. Fusion operators search the four neighboring pixels. The contraction replaces a given pixel with a black one if the neighbors contain more than one black pixel. The expansion, on the other hand, replaces the given pixel with a white one if the neighbors contain more than one white pixel.

Before the genetic operations are performed, an objective image, compared with processed images for fitness evaluation, must be provided as a binary drawing image. Target components in the image are represented with white pixels and the remainders are with black ones.

Genetic Algorithms

Chromosomes of the current GA consist of 44 binary strings, assigned to 12 genotypes as shown in Figure 7.16. Phenotypes corresponding to the genotypes consist of onoff states of the operations mentioned above and parameters for the operations concerning thresholding levels. The minimum thresholding levels on the hue and the brightness coordinates, ranging from 0.0 to 1.0, are encoded with 6 bits. Range of their minimum thresholding levels to the maximum ones is encoded with 4 bits in the same manner. Genotypes of the on-off states are encoded with 3 bits; a decimal value from 0 to 3 is defined as an “off” state of the operation and a value of more

0 – 5	6 – 9	10 – 15	16 – 19	20 – 22	23 – 25
TH (minimum)	TH (range)	TB (minimum)	TB (range)	Toggle of TH	Toggle of SM

26 – 28	29 – 31	32 – 34	35 – 37	38 – 40	41 – 43
Toggle of CF	Toggle of CF	Toggle of EF	Toggle of EF	Toggle of EE	Toggle of RV

Fig. 7.16 Locus of genes defined on a chromosome. Each gene consists of several binary strings and has different length depend on its information. A continuous value was binarized with 4 to 6 strings and a toggle of filtering was encoded with 3 strings

than 4 as a state of "on". Such redundant encoding allows sharp changes, caused by one bit reversion, to be avoided.

Figure 7.17 shows the flow diagram to search for appropriate procedures of the segmentation based on GAs. Conventional genetic operations called "simple GA" are used; crossover at the same points of two neighbor chromosomes, random mutation, and ranking depending on fitness evaluation and selection. At the beginning of the GA operation, chromosomes of a certain population size are generated, initialized with random strings. The crossover occurs at the points of certain string length determined at random, and then, each chromosome is mutated with a certain probability per a string. The each chromosome is interpreted as a sequence of filtering operations and their parameters. Subsequently, a clone of the original image is processed using the each sequence. After the fitness between the objective image

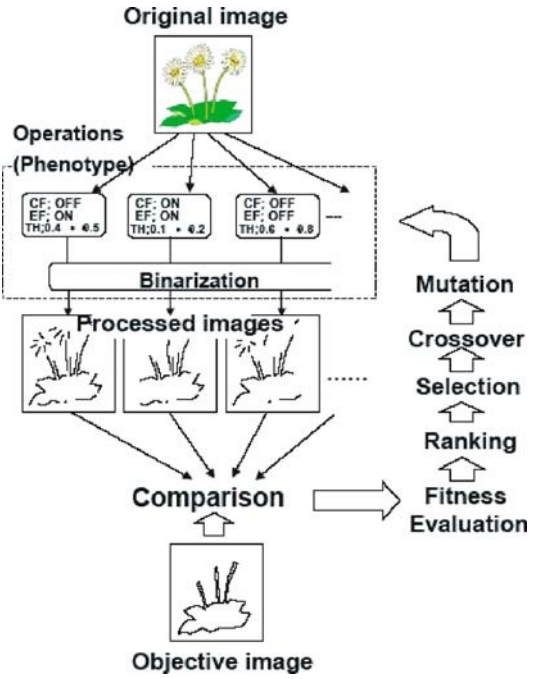


Fig. 7.17 Schematic of the knowledge acquisition system, combining GAs and operations for image

and the processed ones is evaluated, the chromosomes are ranked and selected dependent on the degree of the fitness. The procedure from the crossover to the ranking is performed iteratively until appropriate procedures are obtained. Evaluation and selection play important roles in GAs because they determine the GA's performance of searching for solutions. The function for fitness evaluation is defined by the rate of correspondence between a processed image and an objective one, given as a successfully processed one. In detail, equivalence of each pixel, located in the same coordinate of both images, is verified as shown in the following formula:

$$f = \frac{P_{fit}}{P_{fit} + P_{unfit}}$$

where P_{fit} is the number of pixels equivalent between images and Pun_{fit} is the number in disagreement. After chromosomes are ranked according to their fitness, chromosomes of the population size, with high fitness in the rank, are selected as parents of the next generation.

Programming

Application software, implementing genetic operations described above, has been programmed as a stand-alone application for accessing data files stored in local computers, using the Java language. Java has been used so that the software can remain platform-neutral; the software can work on various operating systems. The software consists of three modules; a module for the genetic operations and image processing, user interfaces for setting parameters and for monitoring passage of processing, and a module for connection with a database. Using a drawing interface to make objective images, users can easily select components in the image to be processed and binarize standard images. Passage of the genetic operations is displayed as phenotypes, processed images with the highest fitness and development of the fitness value. Users can input descriptions on targets segmented, and then, the phenotypes, original images and the descriptions are stored through the connection module.

Database Configuration

The knowledge base consists of three-tiered database architecture as shown in Figure 7.18. The database has been developed using Sybase SQL Anywhere and Symantec db ANYWHERE, middleware for controlling connection from client PCs to the database via the Internet. Since the connection is managed using a user ID and a password embedded in the software, the database is secured from illegal accesses. A table of the database consists of URL of images, description on features of targets segmented, and procedures obtained for acceptable segmentation.

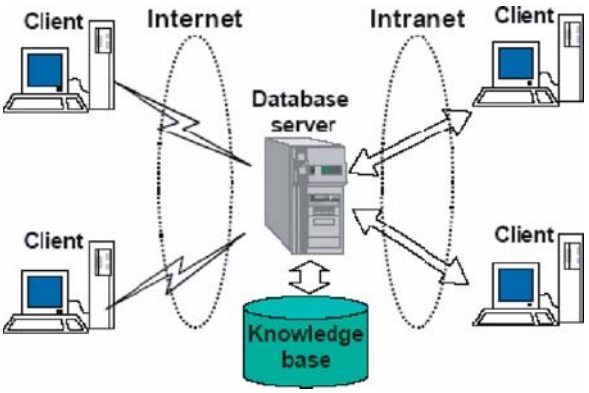


Fig. 7.18 Schematics of the knowledge base for image processing based on three-tiered database

In conventional image processing systems based on knowledge base, all the data for image processing is gotten ready beforehand. On the other hand, the knowledge base has been made without data at the beginning, and is increasing its knowledge as the software is used for processing various images. Figure 7.19 illustrates combination in processing by GA search, represented as the block arrows, with that by the knowledge base, represented as the normal ones.

Specimens

The pictures used are in GIF format with 256 RGB colors and a size of 640*480 pixels. They had been taken with an off-line digital camera and stored in a PC. Their size was decreased to 160*120 pixels by a reduction in the resolution, in order to shorten the time for processing.

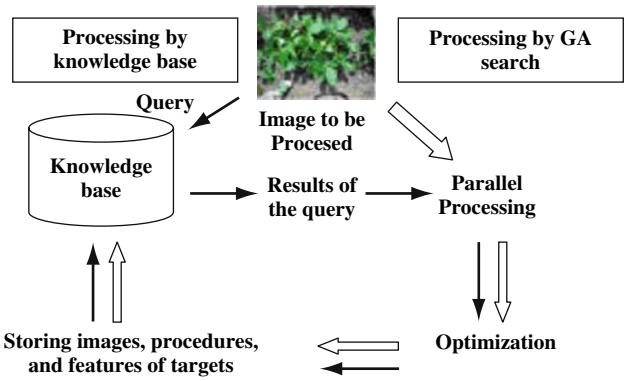


Fig. 7.19 Flow diagram of image processing by two different algorithms

7.4.2 Results and Discussions

Performance of Segmentation

An image of soybean plants has been processed to check the algorithm of the software. The software has been implemented on a PC with a Pentium Pro processor (180MHz) and Microsoft Windows NT version 4.0. The conditions of the genetic operations performed are 50 of the population size and 0.02 of the mutation rate. An objective image, shown in Figure 7.20(b), has been gotten ready beforehand to segment the leaf area of the plants on the original image, Figure 7.20(a). At the beginning of the operations, as shown in Figure 7.20(c), the procedures with the highest fitness in the generation provided an image far from the objective one. The operations performed in the procedures were as follows; TB ranging from 0.62 to 1.0, SM, RV and twice operations of both EF and CF. The highest fitness in each generation, however, steadily increased as the generation advanced. Figure 7.20(d) shows the segmented image closest to the objective one, obtained in the 8th generation. These operations have provided such the acceptable segmentation; TH ranging from 0.21 to 0.41, TB from 0.05 to 1.0, RV and twice iterations of both EF and CF.

Those results show that the genetic operations implemented in the software are available to optimize procedures for image segmentation. To provide a segmented image more acceptable than that, elimination operations based on knowledge of targets, for example, regularity of shape and area, would be necessary. The knowledge obtained by the current software, however, will help to make the elimination process efficient.

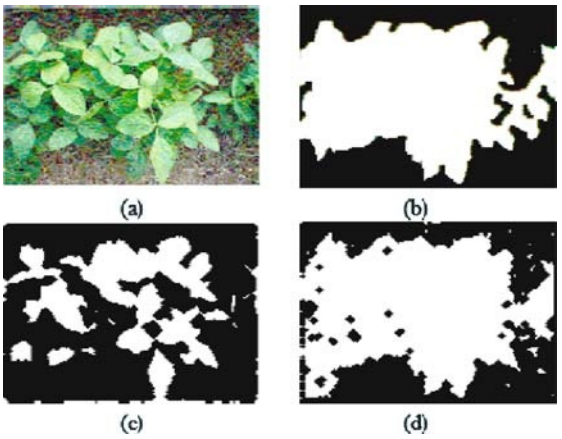


Fig. 7.20 Results of processing an image of soybean plants in the field: (a) an original image; (b) an objective image given; (c) an image processed by the procedure obtained at the first generation; (d) an image processed by the procedure obtained at the 8th generation

Segmentation Using the Knowledge Base

Applicability of the knowledge, procedures for segmentation on various images and targets, stored in the knowledge base, has been investigated. Using the stored procedures, an image of plants in a filed, taken in a situation different from those of the images stored, has been segmented on leaf areas in it. In Figure 7.21, the image labeled O in the left column is the one to be segmented and the image of its right side is an ideal segmented image, gotten ready beforehand. The images, labeled A, B and C in the left column, are stored with the procedures for segmenting the leaf areas in those images. The procedures have been selected from the database, based on similarity of targets in the images. The images of the right column in the figure are clones of the image O, segmented by the each procedure stored with one in the left column. In the image segmented by the procedure of the image A, the group of plants is segmented successfully. On the other hand, the results by the procedures of the image B and C cannot provide successful segmentation of the

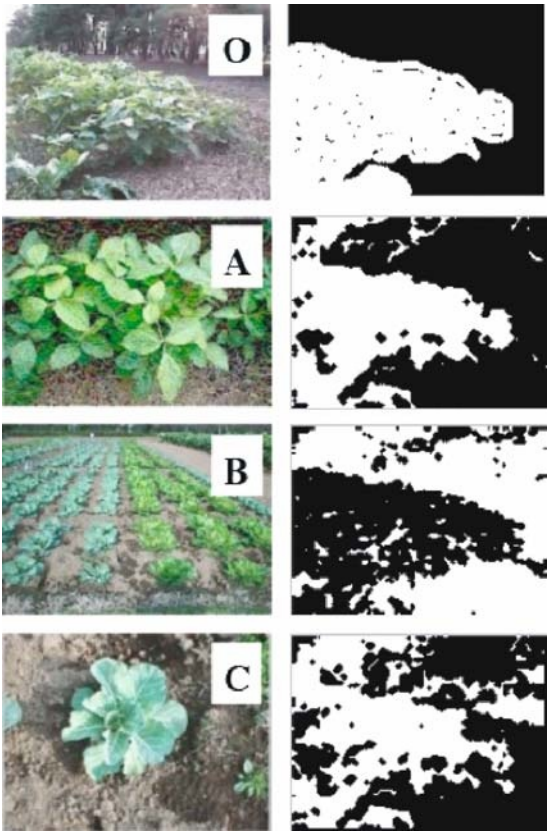


Fig. 7.21 Segmentation by the procedures stored in the knowledge base. The images in the right column are the image O, segmented by the same procedures as those segmenting the plant areas in the images A to C

plants focused. This result shows that the procedure of image A provides the most successful segmentation. Thus, it has been indicated that procedures in the knowledge base have potential to provide acceptable images segmented on various images rapidly.

7.4.3 Concluding Remarks

Thus it is shown that knowledge of the procedures for acceptable segmentation of color images can be easily obtained using genetic algorithms. Using the software developed, those procedures obtained, simultaneously, can be stored in the knowledge base with images before segmented and descriptions of features on targets. Since the procedures can be used for segmenting different images, the knowledge base can be smarter as various images are processed. In addition, there is no limitation on using the database jointly because the software and the database can jointly work through the Internet. Thus, this architecture has an advantage over conventional expert system approaches, implementing all the knowledge for image processing in the system ahead.

7.5 Map Segmentation by Colour Cube Genetic *K*-Mean Clustering

Image segmentation is a low-level image processing task that aims at partitioning an image into homogeneous regions. How region homogeneity is defined depends on the application. A great number of segmentation methods are available in the literature to segment images according to various criteria such as for example grey level, colour, or texture. This task is hard, since the output of an image segmentation algorithm can be fed as input to higher-level processing tasks, such as model-based object recognition systems. Recently, researchers have investigated the application of genetic algorithms into the image segmentation problem. Many general pattern recognition applications of this particular paradigm are available. One reason (among others) for using this kind of approach is mainly related with the GA ability to deal with large, complex search spaces in situations where only minimum knowledge is available about the objective function. For example, most existing image segmentation algorithms have many parameters that need to be adjusted. The corresponding search space is in many situations, quite large and there are complex interactions among parameters, namely if it is required to solve colour image segmentation problems. This led to adopt a GA to determine the parameter set that optimise the output of an existing segmentation algorithm under various conditions of image acquisition. That was the case for the optimisation of the *Phoenix* segmentation algorithm, by genetic algorithms. In another situation wherein GAs may be useful tools, the segmentation problem was formulated upon

textured images as an optimisation problem, and adopted GAs for the clustering of small regions in a feature space, using also *Kohonen's* self-organising maps (SOM). The original image was divided into many small rectangular regions and extracted texture features from the data in each small region by using the two-dimensional autoregressive model (2D-AR), fractal dimension, mean and variance. In other example, a multi-term cost function was defined, which is minimized using a GA-evolved edge configuration. The idea was to solve medical image problems, namely edge detection. In this approach to image segmentation, edge detection is cast as the problem of minimizing an objective cost function over the space of all possible edge configurations and a population of edge images is evolved using specialised operators. Fuzzy GA fitness functions were also considered, mapping a region-based segmentation onto the binary string representing an individual, and evolving a population of possible segmentations. Other implementations include the search of optimal descriptors to represent 3D structures, or the optimisation of parameters in GA hybrid systems - in this last case, for finding the appropriate parameters of recurrent neural networks to segment echo cardiographic images. GA applications within elastic-contour models are also possible to find.

In another very interesting application, the image to be segmented is considered as an artificial environment wherein regions with different characteristics according to the segmentation criterion are as many ecological niches. A GA is then used to evolve a population of chromosomes that are distributed all over this environment. Each chromosome belongs to one out of a number of distinct species. The GA-driven evolution leads distinct species to spread over different niches. Consequently, the distribution of the various species at the end of the run unravels the location of the homogeneous regions on the original image. Because the segmentation progressively emerges as a byproduct of a relaxation process [9] mainly driven by selection, the method has been called selectionist relaxation.

7.5.1 Genetic Clustering in Image Segmentation

Whether the GA is used to search in the parameter space of an existing segmentation algorithm, or in the space of candidate segmentations, an objective fitness function, assigning a score to each segmentation, has to be specified in both cases. However, evaluating a segmentation result is itself a difficult task. To date, no standard evaluation method prevails, and different measures may yield distinct rankings (as an aside note, the present authors are nowadays developing image noise measures by mathematical morphology, allowing for instance, his use in image filtering design by GAs). One possible criterion is to think of homogeneous regions as the result of any appropriate and optimised clustering process, within the image feature space. In the present approach, grey level intensities of RGB image channels are considered as feature vectors, and the k-mean clustering model is then applied as a quantitative

criterion (or GA objective fitness function), for guiding the evolutionary algorithm in his appropriate search. Since the *k*-mean clustering model allows to minimize the internal feature variance of each colour cluster (or the maximisation of the variances between different colour clusters), natural and homogeneous clusters can emerge if the GA is properly coded. In other words, the image segmentation problem is simply reformulated as an unsupervised clustering problem, and genetic algorithms are then used for finding the most appropriate and natural clusters. Since the clustering task can not be successfully applied within the image 2D space itself (e.g., similar pixels can be very far apart) the problem is coded within another space - that one of their colour features - 3D (grey level intensities, for the three channels). By this reformulation, one can in fact guarantee that similar pixels will belong to the same colour cluster.

7.5.2 K-Means Clustering Model

K-Means clustering models were introduced in 1967 by *J. MacQueen*, and they are considered as an unsupervised classification technique. Once the method uses a minimum distance criteria, many authors consider this approach in many ways similar to the k-nearest neighbour rule method (there is also some similarities with the *Kohonen* LVQ method). In *MacQueen* terms however, k stands for the number of clusters searched by the model (and given as an input). All the strategy undergoes the minimisation of the expression (7.1). If one admits the partition of a p-dimensional space into c clusters (c colour classes), where n samples exists (n points characterised by p features), and being Ci each cluster i centre $(i = 1,\ldots,c)$, and Xj representative of each sample j co-ordinates $(j = 1,\ldots,n)$, where u_{ij} represents the hypothetical belonging of sample j into cluster i (i.e., $u_{ij} = 1$ if j belongs to cluster i; $u_{ij} = 0$ if j belongs to any other cluster different from i). Naturally for the present case, the idea is to compute the colour cube partition minimising J by using genetic algorithms. Then J is obtained as in Eq.7.1 ($u_{ij} = 0,1$; $\sum u_{ij} = 1$ and $\sum\sum u_{ij} = n$):

$$\begin{aligned} & u_{ij} \in \bigcup_{cxn}; \\ & C_i = \frac{1}{n} \sum_{j=1}^{n} X_j; \\ & \min J = \sum_{i=1}^{c} \sum_{j=1}^{n} u_{ij} \left\| X_j - C_i \right\|^2 \end{aligned} \tag{7.1}$$

7.5.3 Genetic Implementation

The above minimisation is then based on the different belonging combinations, of all points in the feature space. Naturally that, such task will be simply if the number

of colours in one image to segment is low; however for high number of points in this 3D colour space (i.e., the different number of colours) this minimisation is hard to compute, since the combinatorial search space becomes very large. However, the partition of this 3D histogram into different clusters, must take in account the value of each point (that is, this frequency for a given RGB point). By this, the minimisation suffers a little modification (the method becomes weighted by this frequency f, since the number of colours of any RGB point are an important information in the overall process). The above J expression then becomes (Eq.7.2).

$$J = \sum\sum u_{ij} f_j \left|X_j - C_i\right|^2 \tag{7.2}$$

Another important issue in the GA implementation is the problem genetic coding. In order to do it appropriately, each chromosome codes the binary values of u_{ij}. However, to improve the GA search time and since the number of different colours in one colour image can be high, each 3D feature colour was submitted to a pre-partition. By this pre-procedure, the combinatorial search space is reduced, as also as the number of bits in each GA chromosome. That is, each 3D colour cube (with side 256 - 8 bit images) that could represent up to 2563 colours, was reduced to a maximum of 512 points (i.e., 512 small cubes with side 32). In other words, all RGB points that fall into a small cube are agglomerated, being the new point represented by the centre of this small cube, and his frequency being equal to the sum of all frequencies of those points.

7.5.4 Results and Conclusions

The above strategy was applied in colour maps (214385 different colours - as in Figure 7.22). Since the colour map had roughly six type of colours the aim was to segment it into 6 clusters (6 prominent colours). The number of GA individuals were always 50, generations run= 10000, crossover mutation rate= 0.95, and

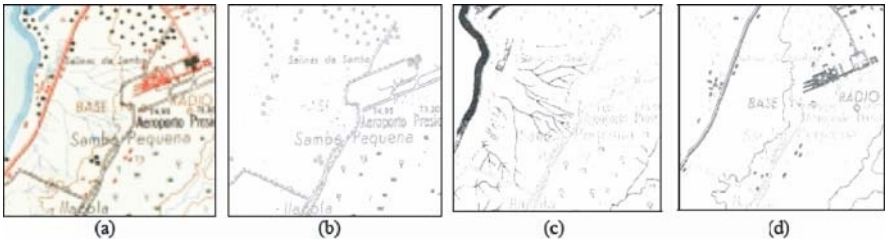


Fig. 7.22 (a) Original Luanda (Angola) Colour Map [500x500 pixels] and (b) the respective 1st colour cluster segmenting train networks (in black), names (in black and dark brown), and parks (in dark green) pointed by the GA; (c) 2nd colour cluster – sea and rivers (blue); (d) Roads and buildings (red) and topological levels (brown). Other examples not shown here, reflect darker borders (red, black and brown) and background (whiter regions)

mutation rate= 0.85. The respective computer time (PENTIUM 166 MHz/32 Mb Ram) was 37.02 minutes. Finally, each string length (a parameter that depends only in the number of different colours present) was 468 bits long. Overall results points to highly satisfactory results namely. There is however some problems with some colour clusters. The main reason is that the problem is by itself difficult (with large combinatorial search spaces), and that the pre-partition tends to reduce the discriminatory power of the overall strategy. This is mainly observed within pixels that form bounds of any important colour object. Image acquisition with low resolutions interpolates somehow their grey level intensities into intermediate values (between inner and outer bounds), and the result (with pre-partition) is significantly altered, since similar pixels can belong to different small cubes (naturally with low probability). However if the number of this kind of pixels is high, the strategy tends to create himself another cluster. On the other hand, and by observing the GA performance (J value) in each generation, it is concluded that similar results can be achieved with half of the generations run, since after 5000 generations J values are increasing very slow.

7.6 Genetic Algorithm-Based Performance Analysis of Self-Excited Induction Generator

A self-excited induction generator (SEIG) is usually utilized to supply electric loads in remote areas not served by the grid, because of its advantages over a synchronous generator. A conventional induction machine can be operated as a self-excited induction generator, if its rotor is externally driven at a suitable speed and sufficient excitation capacitance is provided. The performance characteristics of a self-excited induction generator can be obtained after the determination of two unknown parameters, such as the magnetizing reactance and frequency. Usually, Newton–Raphson method and Nodal-Admittance method are used to determine the generator's unknown parameters. If either of these two methods is used, lengthy mathematical derivations should be carried out to formulate the required equations in a suitable simplified form. The mathematical derivations are carried out by hand, and therefore considerable amounts of time and effort are needed. Moreover, having mistakes in the final form equations is possible. However, mistakes can be avoided if the mathematical derivations are carefully carried out. The Newton–Raphson method may lead to inaccurate results if initial estimates are not very carefully selected. The other limitation of these methods is the difficulty in using them for the analysis of self-excited induction generator feeding a dynamic load. For this type of analysis, more than two unknowns have to be determined, and these parameters cannot be directly determined using either of these methods. This application deals with the implementation of a new approach, based on genetic algorithm, for the performance analysis of self-excited induction generator. Unlike conventional methods of analysis, lengthy algebraic derivations or accurate initial estimates are not required. In addition, the same objective function is to be minimized irrespective of

the unknown parameters. The other important feature of the present approach is the possibility of determining more than two unknown parameters simultaneously. Therefore, it can be used to obtain the performance characteristics of three-phase (or single-phase) self-excited induction generator feeding three-phase (or single-phase) induction motor. To confirm the validity of the proposed approach, simulation results are compared with the corresponding experimental test results.

7.6.1 Modelling of SEIG System

The steady-state per-phase equivalent circuit of a SEIG supplying a balanced RL-load is shown in Figure 7.23. In this model, the normal assumptions are considered: core losses as well as harmonic effects are ignored and all machine parameters, except the magnetizing reactance XM, are assumed constant and independent of magnetic saturation. Using the circuit of Figure 7.23, the loop equation can be given by:

Under steady-state self-excitation, the total impedance ZT must be zero, as the stator current IS should not be zero. ZT can be expressed by:

$$Z_T = \frac{Z_L Z_C}{Z_L + Z_C} + Z_S + \frac{Z_M Z_R}{Z_M + Z_R}$$

where

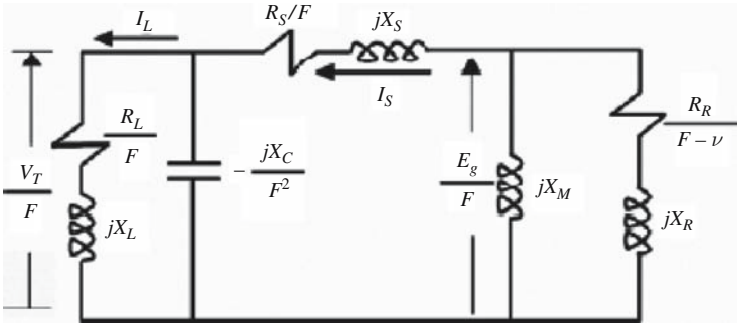


Fig. 7.23 Per-phase equivalent circuit of a three-phase SEIG

$$Z_L = \frac{R_L}{F} + jX_L$$
$$Z_C = \frac{jX_C}{F^2}$$
$$Z_S = \frac{R_S}{F} + jX_S$$
$$Z_M = jX_M$$
$$Z_R = \frac{R_R}{F - \nu} + jX_R$$

The above equation is a nonlinear equation of four variables (F,XM,XC, í). If two of these parameters are given, a numerical technique, like Newton–Raphson method, could be used to find the values of the two unknowns, such as XM and F. From the generator's magnetization curve, which relates Eg/F to XM, the value of Eg/F for the computed XM can be obtained. Using the computed F value, the value of Eg can be determined. With known values of Eg, XM, F,XC, v, RL, XL and the generator's equivalent circuit parameters (RS,RR,XS,XR), the performance characteristics of the generator could be evaluated.

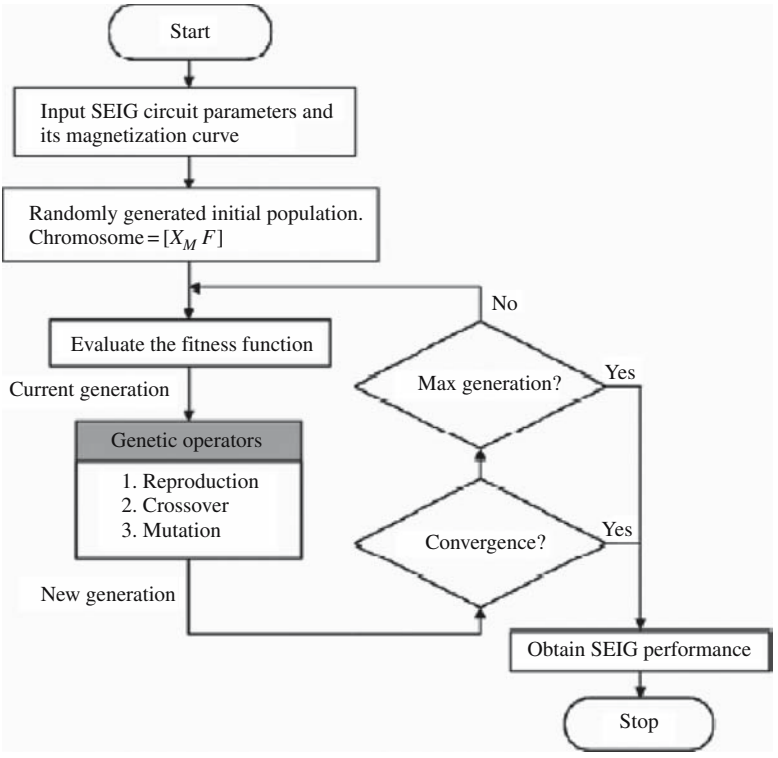


Fig. 7.24 Flowchart of GA optimization system

7.6.2 Genetic Algorithm Optimization

Genetic algorithm (GA) is becoming a popular method for optimization, because it has several advantages over other optimization methods. It is robust, able to find global minimum, and does not require accurate initial estimates. In addition, detailed derivations of analytical equations to reformulate an optimization problem into an appropriate form are not required; GA can be directly implemented to acquire the optimum solution using a certain fittness function. The flowchart describing the GA optimization system implemented in this application is shown in Figure 7.24. The two unknowns XM and F are determined by optimizing (minimizing) the objective function (total impedance equation). Similar optimization procedure is applied to obtain other combinations of two unknown parameters, such as XC and v.

GA optimization is started with a population of N randomly generated individuals representing a set of solutions to the problem. Each individual is composed of the problem's variables, which can be encoded into binary strings called chromosomes. GA works on these chromosomes instead of the variables themselves. The variables can also be represented using real-valued (decimal) encoding. For this type of encoding, GA operates directly on the variables of the optimization problem. In this application, the population size N is chosen to be 70 and the variables in each individual (chromosome) are the unknown parameters: chromosome=[XMF]. These genes (unknowns) are represented using real-valued (decimal) encoding. Higher computational efficiency and greater freedom in selecting genetic operators are among the advantages of this encoding type. After the generation process is completed, stochastic rules called genetic operators are applied to the individuals in a reproduction process, which generates a new and improved generation. This process will be repeated until one individual emerges as the best. The main three genetic operators are reproduction, crossover, and mutation. Reproduction is a probabilistic selection process in which individuals of the old population are selected, according to their fittness, to produce the next generation. The fittness function has to be defined to evaluate how good each individual is. In this application, the objective function (the total impedance equation) is considered as the fittness function. The reproduction process provides copies from the genetically good individuals of the current generation, keeping the population size fixed. Selection alone cannot introduce any new individuals into the population. Therefore, the crossover operator is applied to make individuals exchange information. Crossover randomly chooses two individuals, called parents, to exchange genetic information with each other and generates two new individuals, called children. The crossover operation is repeated 35 (N/2) times, and another 70 children (individuals) will be generated. The children compete with the parents in the reproduction process to form the new generation. As shown in Figure 7.25, the crossover operator can be mathematically described as follows:

If parents are (XM1, F1) and (XM2, F2) then:

$$\text{Child-1}: \begin{cases} X_M = rX_{M1} + (1-r)X_{M2} \\ F = rF_1 + (1-r)F_2 \end{cases}$$

$$\text{Child-2}: \begin{cases} X_M = (1-r)X_{M1} + rX_{M2} \\ F = (1-r)F_1 + rF_2 \end{cases}$$

where r ε (0, 1) is a random number called weighted average operator (crossover rate), and is chosen to be 0.9. High crossover rate is chosen to maintain good genetic information for each child and at the same time keep the crossover process behaving properly. Mutation is the process of random modification of the value of the individual with a small probability. It is not a primary operator, but it prevents premature convergence to local optima. For real-valued encoding, mutation operator can be mathematically described, if X_M and F are the variables in each individual, as:

$$X_M = X_M + (r_1 - 0.5)(2X_{M\ \max})$$
$$F = F + (r_2 - 0.5)(2F_{\max})$$

where r_1, r_2 ε (0, 1) are two random numbers and XM_{max} and F_{max} are maximum changes of X_M and F under mutation. The selected mutation rate is 0.1 ($r_1 = r_2 = 0.1$). In addition to GA operators, the performance of GA optimizer is affected by the termination condition of the GA optimization. To obtain accurate results, the "best" results, an appropriate convergence decision to terminate GA operations is needed. Two termination decisions are considered in this application:

(1) if the fittness function (total impedance equation) value, during 150 generations, reaches a very small value (less that 10–7) and then does not change much (å$< 5 \times 10 - 9$), the optimization process is terminated;
(2) otherwise, the process must be terminated at the maximum number of generations. This number is chosen to be 500.

7.6.3 Results and Discussion

The machine under test is a three-phase, four-pole, 50 Hz, 220 V, 6.6 A, 1.5 kW, delta-connected squirrel cage induction machine, whose per phase equivalent circuit parameters in p.u. are:

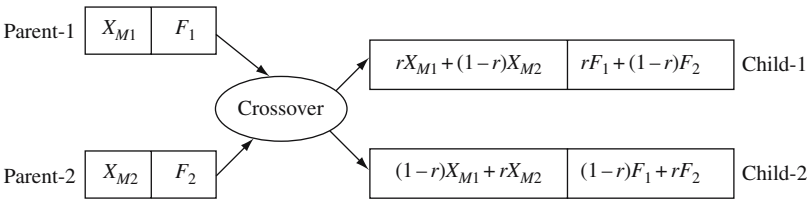


Fig. 7.25 Crossover operator

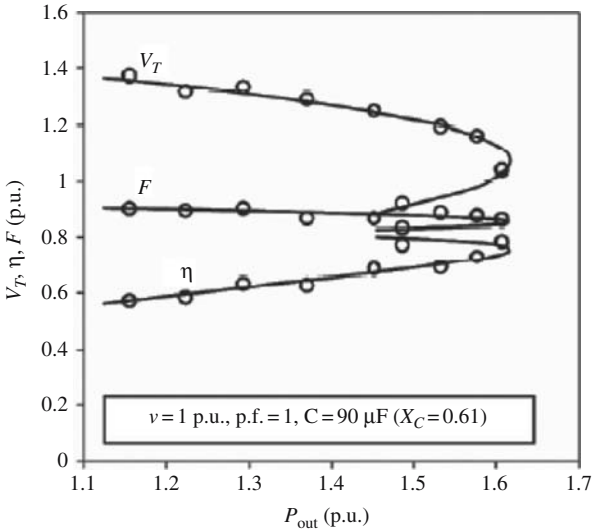


Fig. 7.26 Variations of V_T, c, and F versus P_{out} under constant speed operation

$$R_S = 0.07\ R_R = 0.16\ X_S = 0.22\ X_R = 0.34\ X_{MU} = 2.25$$

The magnetization characteristic of this machine is mathematically expressed by:

$$\frac{E_g}{F} = -0.16X_M^3 + 1.32X_M^2 + 1.31X_M + 1.51$$

where $0 < X_M < 2.25$. To verify the validity for the implementation of GA approach for steady-state analysis of SEIG, experiments are conducted under similar conditions and the simulation results (continuous curves) are compared with their corresponding test results (circle points). Figure 7.26 shows the results for the variations of terminal voltage, frequency, and efficiency with output power for fixed excitation capacitance, at constant speed. It can be noted that the terminal voltage and frequency decrease as the load increases, and the generator's efficiency improves with load. Moreover, it can be observed that the generator becomes unstable when the magnetizing reactance exceeds the unsaturated value: $X_M > X_{MU}$. The results representing the variations of the output power, frequency, and terminal voltage with speed, for constant values of load and excitation capacitance, are given in Figure 7.27. From this figure it can be seen that the terminal voltage, the output frequency, and the output power increase approximately linearly with rotational speed until the unsaturated region is reached.

Figure 7.28 shows the variations of the exciting capacitance and frequency versus load impedance at fixed speed and terminal voltage. As can be observed, the required exciting capacitance decreases as the load impedance increases, whereas the frequency increases with load impedance. Figure 7.29 shows the variations of

the generator's terminal voltage and its rotational speed against the load current, under constant frequency operation. As can be seen, the terminal voltage drops with load. In addition, the rotational speed is increased as load increases to maintain the output frequency fixed. The required excitation capacitance values to obtain constant output voltage for different rotational speeds are shown in Figure 7.30. The variations of the output frequency versus speed are included in this figure. It can be noted that the required excitation capacitance increases as the speed decreases and the output frequency increases approximately linearly with speed. As can be seen in the previous figures, computed results are very close to experimental test results. The results and findings presented in this section verify the validity, accuracy, and feasibility of utilizing the proposed approach for steady-state analysis of self excited induction generators.

7.6.4 Concluding Remarks

In this application, a new approach, based on genetic algorithm optimization procedure, has been implemented successfully for steady-state analysis of self-excited induction generators under different operating conditions. Unlike other reported methods, with the proposed method the desired unknown parameters are obtained without the need for good initial estimates or lengthy algebraic derivations. To confirm the validity of the proposed approach, the simulation results are compared with the corresponding experimental test results. It is found that the results of the proposed technique are in close agreement with the experimental test results.

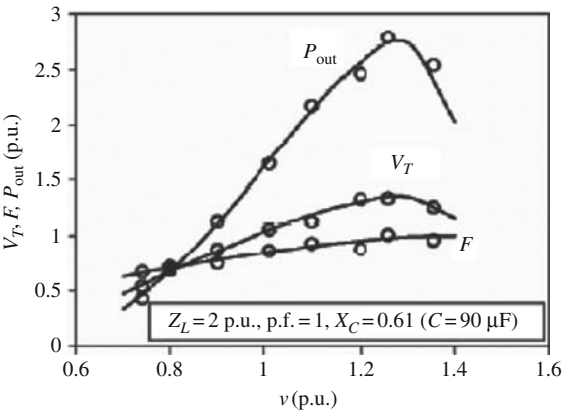


Fig. 7.27 Variations of V_T, P_{out}, and F against v, for fixed values of ZL and C

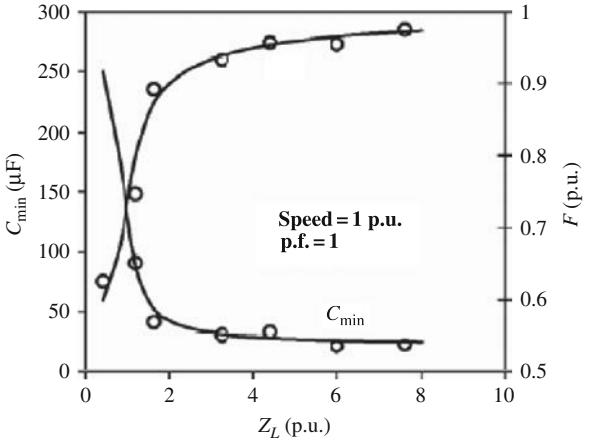


Fig. 7.28 Variations of C_{min} and F with Z_L, under constant v and V_T operation

7.7 Feature Selection for Anns Using Genetic Algorithms in Condition Monitoring

As modern day production plants are expected to run continuously, industry has created a demand for techniques that are capable of recognizing the development of a fault condition within a machine system. Machine Condition Monitoring was developed to meet this need. Research has shown that Artificial Neural Networks (ANNs) are promising solution to several different problem areas; however many of the input features used require a significant computational effort to calculate. A feature

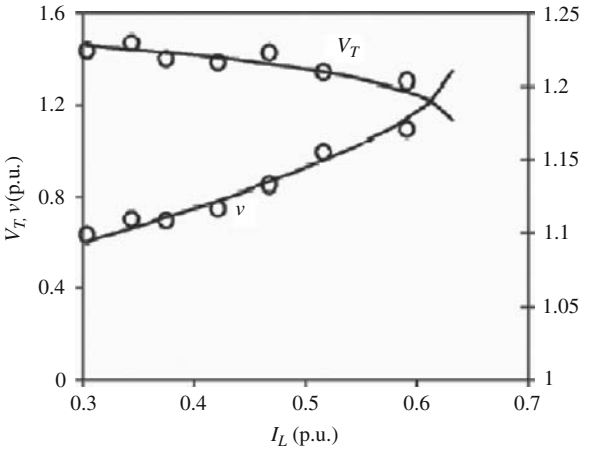


Fig. 7.29 Variations of V_T and v against I_L under constant frequency operation

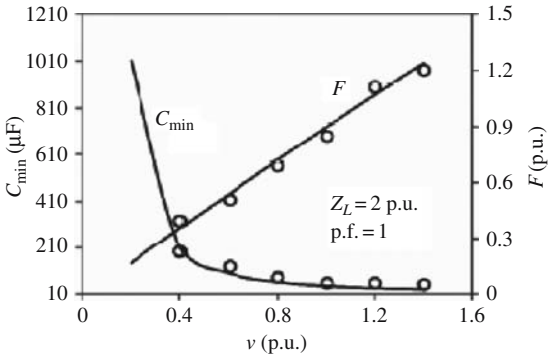


Fig. 7.30 Variations of C_{min} and F with speed, at fixed terminal voltage

selection process using GAs is used in order to isolate features that are provide the most significant information for the neural network, whilst cutting down the number of inputs required for the network. This application is based on experimental results performed on vibration data taken from a small test rig shown in Figure 7.31, which was fitted with a number of interchangeable faulty roller bearings. This is used to simulate the type of problems that can commonly occur in rotating machinery.

Rolling element, or ball bearings, are one of the most common components in modern rotating machinery; being able to detect accurately the existence of a fault in a machine can be of prime importance in certain areas of industry. Six differ-

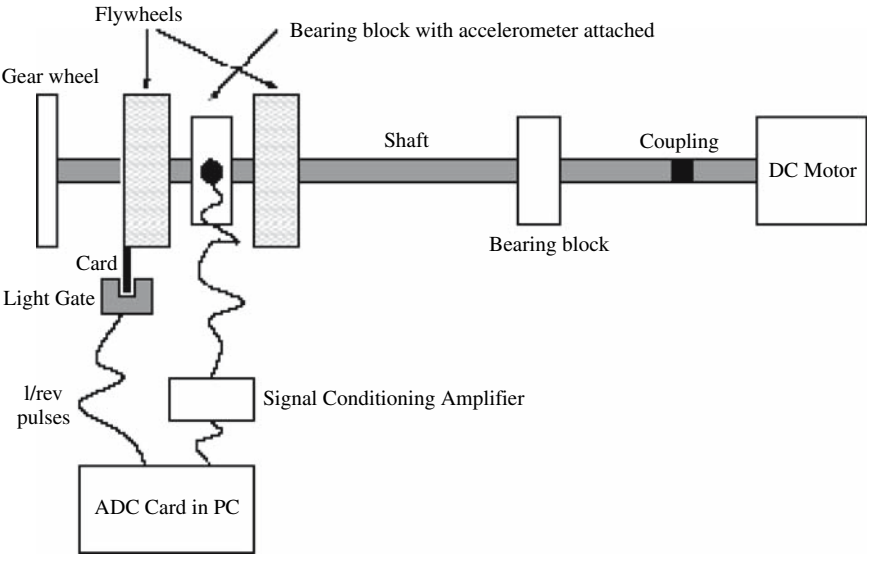


Fig. 7.31 The machine test rig used in experiments

ent conditions are used within the experiments conducted for this application. Two normal conditions exist: one bearing in a brand new condition, while another is a bearing in slightly worn condition. There are four fault conditions:- inner race fault, outer race fault, rolling element fault, and a cage fault.

7.7.1 Signal Acquisition

The signal from the accelerometers was passed through a low pass filter with a cutoff frequency of 18.3 kHz, and was then sampled at a rate of 48 kHz, giving a slight oversampling of the data. This operation was repeated ten times for 16 different speeds. With a total of six different conditions, this gives a total data set of 960 cases, with 160 cases for each condition.

Vibration Datasets

A number of different statistical features were taken based on the moments and cumulants of the vibration data. Higher order spectra have been found to be useful in the identification of different problems in condition monitoring. Using the measured signals, three datasets were prepared. The first dataset contained moments and cumulants of up to order 4 - a matrix of 960 entries. A spectrally based feature set was created. For each of the two channels sampled, a 32-point FFT of the raw data was carried out, and 33 values were obtained for each channel. These were then stored as a column vector of 66 values, which was used as the input data set for the given data sample. The full input data set formed a 66×960 data set. Combining these two datasets gave a larger third set with a total of 156 features (156×960). This dataset, and the other two smaller feature sets were all normalised before training.

7.7.2 Neural Networks

The MLP used in this application consists of one hidden layer and an output layer, the hidden layer having a logistic activation function, whilst the output layer uses a linear activation function. The size of the hidden layer is determined by the genetic algorithm itself during training. This allows training to proceed at a faster rate than an exhaustive training process that checks different sizes of the first layer. The size of the second layer is determined by the number of outputs required. This is set at six neurons for the particular application. Training of the network is carried out using a standard back-propagation algorithm, and the network is trained for 350 epochs, using 40% of the data set as training data, and the remaining 60% as the test and validation set.

7.7.3 Genetic Algorithms

GAs have been gaining popularity in a variety of applications which require global optimisation of a solution. The prime component of a genetic algorithm is the genome. The genome is an encoded set of instructions which the genetic algorithm will use to construct a new model or function (in this case the inputs to a neural network). The best type of encoding is very much problem dependent, and may require some form of combination of two or more encoding types (binary, real numbers, etc) in order to get the optimum results. The GA is allowed to select subsets of various sizes in order to determine the optimum combination and number of inputs to the network. The emphasis in using the genetic algorithm for feature selection is to reduce the computational load on the training system while still allowing near optimal results to be found relatively quickly.

Feature Selection & Encoding

Feature selection of the GA is controlled through the values contained within the genome generated by the GA. On being passed a genome with $(N+1)$ values to be tested, the first N values are used to determine which rows are selected as a subset from the input feature set matrix. Rows corresponding to the numbers contained within the genome are copied into a new matrix containing N rows. The last value of the genome determines the number of neurons present in layer 1 of the network. For this particular application, a simple real number based genome string was used. For a training run requiring N different inputs to be selected as a subset of Q possible inputs, the genome string would consist of $(N+1)$ real numbers. The maximum number of neurons permissible in the first layer is defined as S. Each of the first N numbers (x) in the genome is constrained as $0 \leq x \leq (Q-1)$, whilst the last number, (x), is constrained to lie within the bounds $1 \leq x \leq S$. This means that any mutation that occurs will be bounded within the limits set at the definition of the genome. The classification performance of the trained network using the whole dataset was returned to the GA as the value of the fitness function. The GA uses a population size of 10 individuals, starting with randomly generated genomes. The probability of mutation was set to 0.2, whilst the probability of crossover was set to 0.75. An elitist population model is used, meaning that the best individual in the previous population is kept in the next population, and preventing the performance of the GA worsening as the number of generations increase.

7.7.4 Training and Simulation

Training was carried out using three data sets; One feature set comprised all the statistically based features (90 features). The set of 66 spectral features was used as an individual case, and this dataset was combined with all the statistical feature sets

Table 7.3 Performance of straight ANNs using different feature sets

Data Set	No. Inputs	No. Neurons	Classification Success (%)	False Alarm Rate (%)	Fault Not Recognised (%)
All Statistics	90	7	88.1	0.2	7.4
Spectral Data	66	8	97.0	0.2	2.2
All Data	156	6	91.1	0.1	10

to form an input feature set of 156 inputs. Each feature set contained a total of 960 cases. Using the genetic algorithm running for a total of 40 generations, each containing 10 members (meaning the training of 400 neural networks), eight separate cases were tested using various numbers of inputs, varying from five to twelve. As a comparison, a neural network was trained using each feature set. These were trained for a total of 350 epochs, and allowed to choose the best size of intermediate layer between 2 and 15 neurons.

7.7.5 Results

ANN

Table 7.3 shows a summary of results for all three feature sets used. The "No Neurons" quoted in the second column is the number of neurons used in the hidden layer of the best network in each training run. "Classification success" represents the percentage success rate of the ANN using the complete dataset, which includes both training and test data. The "false alarm" rate details the number of "normal" conditions that were misclassified as alarm conditions, expressed as a percentage of the total dataset. The "fault not recognised" category details the number of fault conditions that were classified as normal, again expressed as a percentage of the total dataset. As can be seen, all the feature sets have a performance greater than 88%, with two cases in excess of 91%; the spectral feature set gives the best performance, at 97% for the overall data set. While this is a comparatively small feature set containing 66 features, the spectral content of the data is ideally suited to recognising several of the periodic type faults that are generated by the different conditions. The aggregate of the false alarm rate and fault not recognised rates is also the lowest of all the different feature sets.

The feature set containing all the data has a larger number of features than any of the other sets, and the drop in performance may be due to the limited amount of time that the training algorithm is allowed to arrive at a result. It may be that due to the large number of features the training algorithm is unable to arrive at a better solution before the training cycle is stopped.

Genetic Algorithm with ANN After 40 Generations

Table 7.4 shows the performance of the different feature sets after running under the GA for 40 generations. All of the datasets have their best performance in excess of 97.5%. The feature set using all the available training data has managed to achieve an accuracy of 100%, indicating accurate classification. This is achieved using only six inputs out of the possible 156. Using 9 neurons in the hidden layer, a relatively small network has been created that fulfils the criteria set earlier on. A network of this size would be ideal for a real time implementation on a small chip or microcontroller.

7.7.6 Concluding Remarks

The use of the Genetic Algorithm allows feature selection to be carried out in an automatic manner, meaning that input combinations can be selected without the need for human intervention. This technique offers great potential for use in a condition monitoring environment, where there are often hundreds and even thousands of different measurements available to a monitoring system, and selection of the most relevant features is often difficult. It has been shown that the Genetic algorithm is capable of selecting a subset of 6 inputs from a set of 156 features that allow the ANN to perform with 100% accuracy. The performance of networks trained using the feature selection was consistently higher than those trained without feature selection.

7.8 A Genetic Algorithm Approach to Scheduling Communications for a Class of Parallel Space-Time Adaptive Processing Algorithms

For an application on a parallel and embedded system to achieve required performance given tight system constraints, it is important to efficiently map the tasks and/or data of the application onto the processors to reduce inter-processor com-

Table 7.4 Comparison between standalone ANN and GA with ANN after 40 generations, for all three data sets

Data Set	Straight ANN			GA with Best ANN			GA with ANN	
	No. I/Ps	No. Hid.	Perf. %	No. I/Ps	No. Hid.	Perf. %	Mean Perf. %	Perf. Range
All Statistics	90	7	88.1	7	14	97.7	97.1	96.0–97.7
Spectral Data	66	8	97.0	6	11	99.8	99.2	97.3–99.8
All Data	156	6	91.1	6	9	100	98.3	95.0–100

munication traffic. In addition to mapping tasks efficiently, it is also important to schedule the communication of messages in a manner that minimizes network contention so as to achieve the smallest possible communication time. Mapping and scheduling can both – either independently or in combination – be cast as optimization problems, and optimizing mapping and scheduling objectives can be critical to the performance of the overall system. For parallel and embedded systems, great significance is placed on minimizing execution time (which includes both computation and communication components) and/or maximizing throughput. This application involves optimizing the scheduling of messages for a class of radar signal processing techniques known as space-time adaptive processing (STAP) on a parallel and embedded system. A genetic algorithm (GA) based approach for solving the message-scheduling problem for the class of parallel STAP algorithms is proposed and preliminary results are provided. The GA-based optimization is performed offline, and the results of this optimization are static schedules for each compute node in the parallel system. These static schedules are then used within the on-line parallel STAP implementation. The results of the study show that significant improvement in communication time performance are possible using the proposed approach for scheduling. Performance of the schedules were evaluated using a RACEway network simulator.

7.8.1 Overview of Parallel STAP

STAP is an adaptive signal processing method that simultaneously combines the signals received from multiple elements of an antenna array (the spatial domain) and from multiple pulses (the temporal domain) of a coherent processing interval. The focus of this research assumes STAP is implemented using an element-space post-Doppler partially adaptive algorithm. Algorithms belonging to the class of element-space post-Doppler STAP perform filtering on the data along the pulse dimension, referred to as Doppler filtering, for each channel prior to adaptive filtering. After Doppler filtering, an adaptive weight problem is solved for each range and pulse data vector. The parallel computer under investigation for this application is the Mercury RACE multicomputer. The RACE multicomputer consists of a scalable network of compute nodes (CNs), as well as various high-speed I/O devices, all interconnected by Mercury's RACEway interconnection fabric. A high-level diagram of a 16-CN RACEway topology is illustrated in Figure 7.32. The interconnection fabric is configured in a fat-tree architecture and is a circuit switched network. The RACEway interconnection fabric is composed of a network of crossbar switches and provides high-speed data communication between different CNs. The Mercury multicomputer can support a heterogeneous collection of CNs (e.g., SHARC and PowerPCs).

Achieving real-time performance requirements for STAP algorithms on a parallel embedded system like the Mercury multicomputer largely depends on two major issues. First is determining the best method for distributing the 3-D STAP

data cube across CNs of the multiprocessor system (i.e., the mapping strategy). Second is determining the scheduling of communications between phases of computation. In general, STAP algorithms contain three phases of processing, one for each dimension of the data cube (i.e., range, pulse, channel). During each phase of processing, the vectors along the dimension of interest are distributed as equally as possible among the processors for processing in parallel. An approach to data set partitioning in STAP applications is to partition the data cube into sub-cube bars. Each sub-cube bar is composed of partial data samples from two dimensions while preserving one whole dimension for processing. The application here assumes a sub-cube bar partitioning of the STAP data cube. Figure 7.33 shows an example of how sub-cube partitioning is applied to partition a 3-D data cube across 12 CNs.

During phases of data redistribution (i.e., communication) between computational phases, the number of required communications and the communication pattern among the CNs is dependant upon how the data cube is mapped onto the CNs. For example, in Figure 7.33 (a) the mapping of sub-cube bars to CNs dictates that after range processing, CN 1 must transfer portions of it data sub-cube bar to CNs 2, 3, and 4. (Each of the other CNs, likewise, is required to send portions of their sub-cube bar to CNs on the same row.) The scheduling (i.e., ordering) of outgoing messages at each CN impacts the resulting communication time. For example, in Figure 7.33(a) CN 1 could order its outgoing messages according to one of $3! = 6$ permutations. Similarly, a scheduling of outgoing messages must be defined for each CN. Improper schedule selection can result in excessive network contention and thereby increase the time to perform all communications between processing phases. The focus in this application is on optimization of message scheduling, for a fixed mapping, using a genetic algorithm methodology.

7.8.2 Genetic Algorithm Methodology

A GA is a population-based model that uses selection and recombination operators to generate new sample points in the solution space. Typically, a GA is composed of

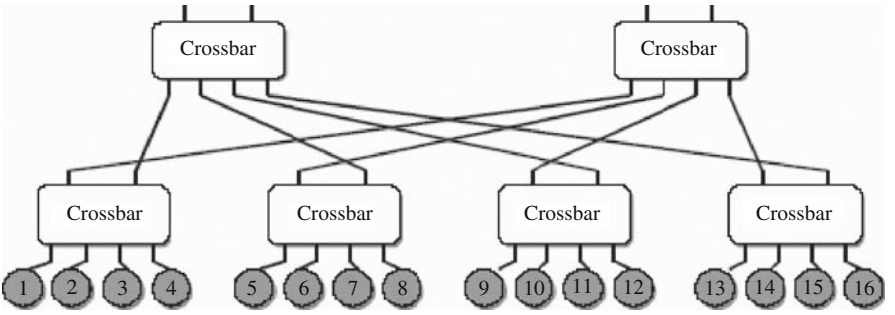


Fig. 7.32 Mercury RACE Fat-Tree Architecture configured with 16 CNs

two main components, which are problem dependent: the *encoding problem* and the *evaluation function*. The *encoding problem* involves generating an encoding scheme to represent the possible solutions to the optimization problem. In this research, a candidate solution (i.e., a chromosome) is encoded to represent valid message schedules for all of the CNs. The *evaluation function* measures the quality of a particular solution. Each chromosome is associated with a fitness value, which in this case is the completion time of the schedule represented by the given chromosome. For this research, the smallest fitness value represents the better solution. The "fitness" of a candidate is calculated here based on its simulated performance. The simulation tool was used earlier to measure the "fitness" (i.e., the completion time) of the schedule of messages represented by each chromosome. Chromosomes evolve through successive iterations, called generations. To create the next generation, new chromosomes, called offspring, are formed by

(a) merging two chromosomes from the current population together using a crossover operator or
(b) modifying a chromosome using a mutation operator.

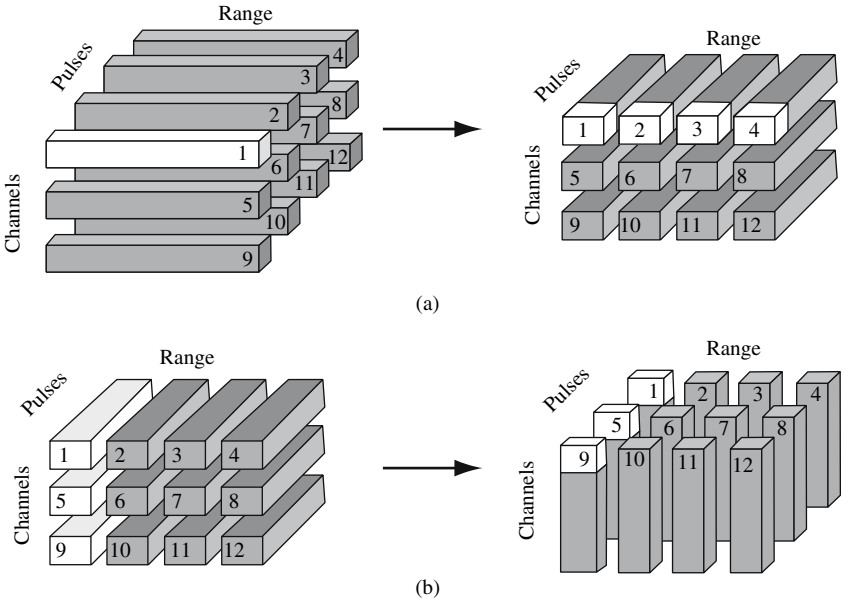


Fig. 7.33 Illustration of the sub-cube bar mapping technique for the case of 12 CNs. The mapping of the sub-cube bars to CNs defines the required data communications. (a) Example illustration of the communication requirements from CN 1 to the other CNs (2, 3, and 4) after completion of the range processing and prior to Doppler processing. (b) Example illustration of the communication requirements from CN 1 to other CNs (5 and 9) after the completion of Doppler processing and prior to adaptive weight processing

Crossover, the main genetic operator, generates valid offspring by combining features of two parent chromosomes. Chromosomes are combined together at a defined crossover rate, which is defined as the ratio of the number of offspring produced in each generation to the population size.

Mutation, a background operator, produces spontaneous random changes in various chromosomes. Mutation serves the critical role of either replacing the chromosomes lost from the population during the selection process or introducing new chromosomes that were not present in the initial population. The mutation rate controls the rate at which new chromosomes are introduced into the population. In this application, results are based on the implementation of a position-based crossover operator and an insertion mutation operator.

Selection is the process of keeping and eliminating chromosomes in the population based on their relative quality or fitness. In most practices, a roulette wheel approach, either rank-based or value-based, is adopted as the selection procedure. In this application, a ranked-based selection scheme is used. Advantages of rank-based fitness assignment is that it provides uniform scaling across chromosomes in the population and is less sensitive to probability-based selections.

7.8.3 Numerical Results

In the experiments reported in this section, it is assumed that the Mercury multicomputer is configured with 32 PowerPC compute nodes. For range processing, Doppler filtering, and adaptive weight computation, the 3-D STAP data cube is mapped onto the 32 processing elements based on an 8×4 process set (i.e., 8 rows and 4 columns). The strategy implemented for CN assignment in a process set is raster-order from left-to-right starting with row one and column one for all process sets. (The process sets not only define the allocation of the CNs to the data but also the required data transfers during phases of data redistribution.) The STAP data cube consists of 240 range bins, 32 pulses, and 16 antenna elements. For each genetic-based scenario, 40 random schedules were generated for the initial population. The poorest 20 schedules were eliminated from the initial population, and the GA population size was kept a constant 20. The recombination operators included a position-based crossover algorithm and an insertion mutation algorithm. A ranked-based selection scheme was assumed with the angle ratio of sectors on the roulette wheel for two adjacently ranked chromosomes to be $1 + 1/\mathrm{P}$, where P is the population size. The stopping criteria were: (1) the number of generations reached 500; (2) the current population converged (i.e., all the chromosomes have the same fitness value); or (3) the current best solution had not improved in the last 150 generations. Figure 7.34 shows the simulated completion time for three genetic-based message scheduling scenarios for the data transfers required between range and Doppler processing phases. Figure 7.35 illustrates the simulated completion time for the same three genetic-based message scheduling scenarios for the data transfers required between Doppler and adaptive weight processing phases. In

the first genetic scenario (GA 1), the crossover rate (Pxover) is 20% and the mutation rate (Pmut) is 4%. For GA 2, Pxover is 50% and Pmut is 10%. For GA 3, Pxover is 90% and Pmut is 50%. Figures 7.34 and 7.35 provide preliminary indication that for a fixed mapping the genetic-algorithm based heuristic is capable of improving the scheduling of messages, thus providing improved performance. All three genetic-based scenarios improve the completion time for both communication phases. In each phase, GA 2 records the best schedule of messages (i.e., the smallest completion time).

7.8.4 Concluding Remarks

In conclusion, preliminary data demonstrates that off-line GA-based message scheduling optimization can provide improved performance in a parallel system. Future expansion will be conducted to more completely study the effect of changing parameters of the GA, including crossover and mutation rates as well as the methods used for crossover and mutation. Finally, future studies will be conducted to determine the performance improvement between a randomly selected scheduling solution and the one determined by the GA. In Figures 7.34 and 7.35, the improvements shown are conservative in the sense that the initial generations' performance on the plots represents the best of 40 randomly generated chromosomes (i.e., solutions). It will be interesting to determine improvements of the GA solutions with respect to the "average" and "worst" randomly generated solutions in the initial population.

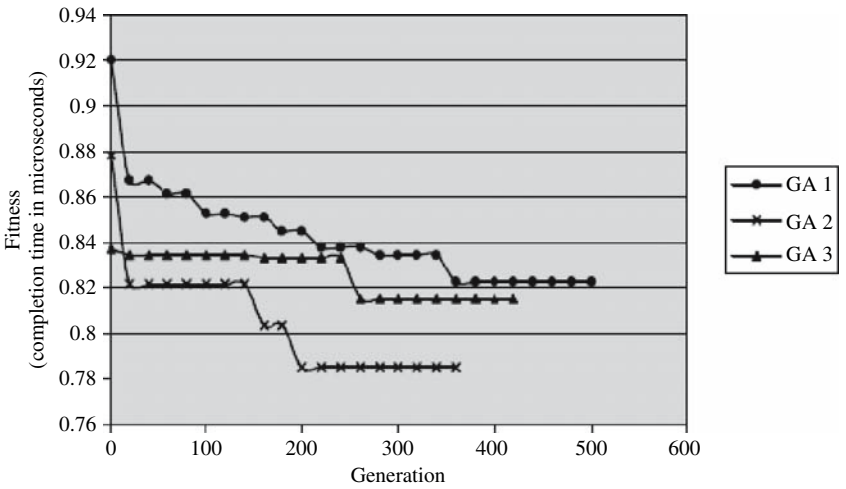


Fig. 7.34 Simulated completion time of the communication requirements for data redistribution after range processing and prior to Doppler processing for the parameters. For GA 1, the crossover rate (Pxover) = 20% and the mutation rate (Pmut) = 4%. For GA 2, Pxover = 50% and Pmut = 10%. For GA 3, Pxover = 90% and Pmut = 50%

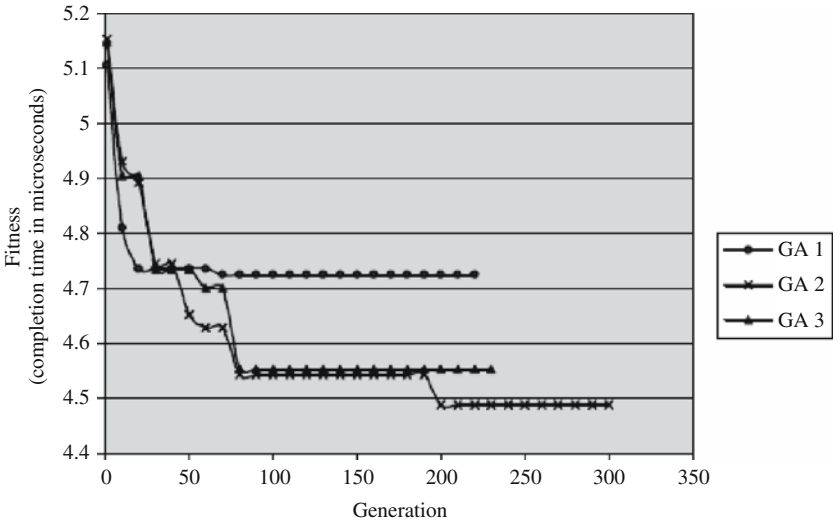


Fig. 7.35 Simulated completion time of the communication requirements for data redistribution after Doppler processing and prior to adaptive weight computation for the parameters. For GA 1, the crossover rate (Pxover) = 20% and the mutation rate (Pmut) = 4%. For GA 2, Pxover = 50% and Pmut = 10%. For GA 3, Pxover = 90% and Pmut = 50%

7.9 A Multi-Objective Genetic Algorithm for on-Chip Real-Time Adaptation of a Multi-Carrier Based Telecommunications Receiver

Multi-carrier code division multiple access (MCCDMA) is one of the most attractive wireless protocols for broadband multimedia communication system. It combines the advantages of orthogonal frequency division multiplexing (OFDM) and code division multiple access (CDMA) to produce a spectrally efficient multi-user access system. The combined requirement for high performance and power is the critical design issue for these wireless systems. These systems must have high performance (in terms of smaller error) at their outputs as they operate in changing environments such as indoor/outdoor, stationary/moving, etc. This implies changes in data rate and bit error rate along with changing bandwidth and other channel parameters such as delay spread. On the other hand, the portability requirement of these systems is mainly responsible for the need for low power consumption. In a MC-CDMA receiver, Fast Fourier Transform (FFT) block is one of the most power consuming units. One way to reduce the power consumption here is by dynamically reducing the complexity of the receiver architecture in real time as per the changing channel requirements such as the delay spread, signal to noise ratio (SNR), bandwidth and bit error rate. It is therefore important to design systems which can adapt their operations instead of being designed for the worst case scenario.

This application presents a multi-objective algorithm for on-line adaptation of a MC-CDMA receiver. A specially tailored GA is developed in order to adapt the complete receiver while dynamically optimizing the critical FFT section of the receiver for both error value and power consumption. The results obtained, through evaluation within complete receiver architecture, demonstrate that the algorithm can find results optimized for both objectives. Results also show that there are significant reductions in error value and power consumption as compared to the reference solution. The application presents the analysis of the impact of word length optimization of fixed-point Fast Fourier Transform (FFT) coefficients on error and power consumption using a non-dominated sorting based GA (NSGA). GA is used in this optimization because it is a class of evolutionary algorithm that can be used for the discovery of near-optimal solutions to multi-modal problems. It makes use of population of solutions, enabling simultaneous discovery of multiple solutions. The application investigates the possibility to find a solution for the FFT coefficients which have optimum performance in terms of error and power consumption as compared to the reference solution. A specific issue of concern in this research would be targeting on-chip optimization of a complete wireless MC-CDMA receiver.

7.9.1 MC-CDMA Receiver

A MC-CDMA receiver consists of two main blocks, an FFT block to demodulate the OFDM signals and a combiner block which equalizes the signal and separates out the coded users. The FFT processor is one of the blocks which contribute the most power consumption. Power consumption of FFT processor depends on the size of the word length of the data and the FFT coefficients. The larger word length means the higher power consumption which is due to more switching activities. On the hands, larger word length means less error and higher signal to noise ratio (SNR). The FFT processor used in this application is based on the radix-4 single-path delay commutator (R4SDC) architecture. This architecture is chosen because it has tremendous saving in hardware and power consumption for real-time applications.

7.9.2 Multi-objective Genetic Algorithm (GA)

Multi-objective GA is crucial in order to optimize multiple fitness measures in many real world problems with multiple conflicting objectives. In multiobjective optimization, there is a possibility that more than one optimal solution is obtained. A set of optimal solutions is called a Pareto-optimal set. This is the main difference between multi-objective optimization and single-objective optimization. In this ap-

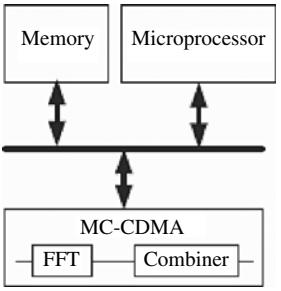


Fig. 7.36 Block diagram of the system

plication, a NSGA is used to optimize the word length of the fixed-point FFT coefficients for two objectives.

(1) Coefficients: they should have smallest total error as possible and,
(2) Power consumption: the complete receiver must have the lowest power consumption.

These two objectives conflict with each other, as an increase in the word length will reduce the error but increases the power consumption.

System Description

Figure 7.36 shows the block diagram of the targeted system which consists of three main blocks namely, memory, microprocessor, and reconfigurable MCCDMA receiver. The choice of the microprocessor is an OpenRISC 1200. This is a common MIPS-based architecture which has been used for a spectrum of implementations at a variety of price/performance levels for a range of industrial telecommunication applications. The main function of the microprocessor is executing the adaptation algorithm to find solutions of the optimized FFT coefficients for a specific word length. This is then propagated through the complete receiver. The memory is used to store the adaptation program and the chromosomes.

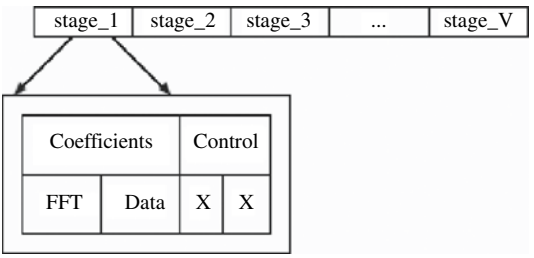


Fig. 7.37 Chromosome structure

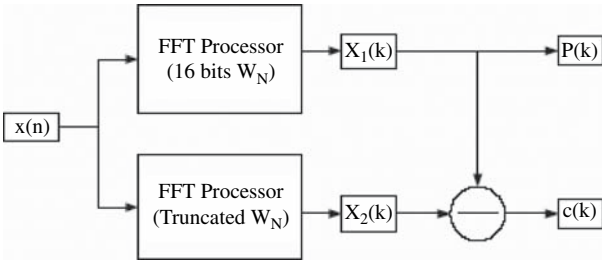


Fig. 7.38 Methodology to evaluate error fitness

Chromosome

Figure 7.37 displays the chromosome representation of the coefficients for the whole FFT processor. Each stage of the chromosome contains 2 fields of information: coefficients and control. The coefficients field stores the real and imaginary parts of the FFT coefficients (twiddle factor), WN and data coefficients, x(n). First stage is the only stage which has both WN and x(n) coefficients. The other stages only have the FFT coefficients except for the final stage. All the coefficients are initially 16 bits in length. The control bits are used to select either FFT coefficients optimization or data optimization or both of these. The length of the chromosome depends on the size of the FFT. For an example, a 16-point FFT will consist of 16 FFT coefficients and 16 data coefficients in the first stage. The initial data and FFT coefficients are obtained from the MATLAB FFT procedure and then are represented as 16-bit numbers. A population size of 50 is chosen through experimentation as it provides sufficient solution diversity.

Error Fitness Evaluation

Figure 7.38 illustrates the methodology used to evaluate the error fitness. Initially, with a sequence of input data, x(n) and 16-bit FFT coefficients, the FFT processor calculates the outputs, $X_1(k)$. Next, with the same x(n) and with the optimized FFT coefficients, the FFT processor again calculates the outputs, $X_2(k)$. Both outputs are then compared for error calculations, e(k). The corresponding SNR(k) for all the FFT outputs are then calculated using the formula in (1).

$$SNR = 10\log\left[\frac{{R_{16}}^2 + {I_{16}}^2}{(R_{16} - R_{wl})^2 + (I_{16} - I_{wl})^2}\right] \tag{7.3}$$

where R_{16}, I_{16}, R_{wl} and I_{wl} are the real and imaginary parts of the FFT output before and after optimization respectively. Next, each SNR(k) value is compared with a target, SNRT. If the former is lower than the latter, an error will be calculated as the difference between the latter and the former. Finally, the total error is calculated by summing the all individual errors.

Table 7.5 Error and switching activity of reference solution

Word length	Switching activity	Total (dB)	Error
10 bits	404	1630	
11 bits	462	1195	
12 bits	502	711	

Power Fitness Evaluation

Power evaluation is performed by calculating the sum of switching activity based on hamming distance in the FFT coefficients using a specific word length. It can be shown that, switching power, P_{sw}, is the main source of power consumption in a typical CMOS logic gate. Equation (2) illustrates how switching power is calculated.

$$P_{SW} = \frac{1}{2} k C_{load} {V_{dd}}^2 f \tag{7.4}$$

V_{dd} is the supply voltage, f is the clock frequency, C_{load} is the load capacitance of the gate, k is the switching activity factor which is defined as the average number of times the gate makes an active transition in a single clock cycle. If, C_{load}, V_{dd} and f are constants, then P_{SW} will be directly proportional to the k.

7.9.3 Results

In this application, the multi-objective Genetic Algorithm is used to adapt the FFT coefficients in the first stage of the FFT processor at word lengths of 10, 11 and 12 bits. The results are compared with a reference solution in terms of switching activity and total error. The reference solution is obtained by truncating all the undesired bits of the FFT coefficients. This is an easier and less complex method to reduce the word length. Table 7.5 displays the figures of total error and switching activity for the reference solution. These figures are obtained using error fitness and power fitness evaluations as explained in sections 3.3 and 3.4 respectively with the chosen SNRT value of 80 dB. The table shows that as the word length increases, the switching activity increases and the total error decreases.

Figure 7.39 shows the best GA search results for the 10-bit word length from five runs. In this experiment, the maximum limit for the switching activity and total error are set to 404 and 1630 dB respectively according to the values indicated in Table 7.5. The figure shows that the GA can find many solutions which meet both objectives.

Table 7.6 lists the Pareto-optimal set for this search which includes the number of generation in which the solution is found, the reduction in switching activity and

Table 7.6 Pareto-optimal set for 10-bit optimization

gen	Switching Activity	Reduction (%)	Total (dB)	Error Reduction (%)
1361	384	5	1343	18
1352	378	6	1354	17
1353	374	7	1362	16
1378	368	9	1392	15
2226	360	11	1456	11

total error. The maximum reduction in switching activity and total errors are 11% and 18% respectively.

Figure 7.40 illustrates the best GA search result for the second experiment on optimization with 11-bit word length obtained after five runs. The maximum limits for the switching activity and total error are set to 462 and 1195 dB respectively based on the values in Table 7.5. The figure shows that the GA is also able to find many solutions which meet both objectives. One of the solutions from the Pareto-optimal set which is found in the 945th generation has the figures for switching activity and total error of 416 and 1028 dB respectively. This implies that the solution is 10% and 14% better than the reference solution in terms of switching activity and total error respectively.

The GA is also able to find many solutions which meet both objectives in the third experiment on optimization for 12-bit word length. One of the solutions from the Pareto-optimal set which is found in the 709th generation has the switching activity and total error figures of 422 and 632 dB respectively. This implies that the solution is 11% and 16% better than the reference solution in terms of switching activity and total error respectively.

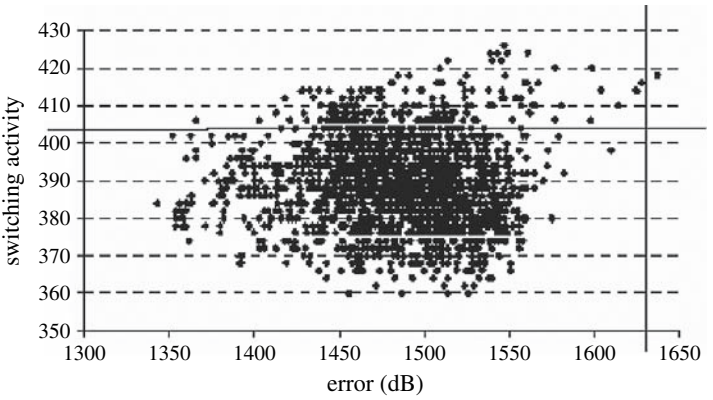


Fig. 7.39 GA search results for 10-bit word length

7.9.4 Concluding Remarks

In this application, a multi-objective Genetic Algorithm based on non-dominated sorting for on-chip real-time adaptation of a multi-carrier receiver used in numerous telecommunication applications was demonstrated. Although the adaptation targets the Fast Fourier section (as this is the most critical block within the receiver), the evaluation is performed on the complete receiver architecture. Results show that the algorithm can find solutions which have significant reductions in error and switching capacitance compared to a reference solution.

7.10 A VLSI Implementation of an Analog Neural Network Suited for Genetic Algorithms

Artificial neural networks are generally accepted as a good solution for problems like pattern matching etc. Despite being well suited for a parallel implementation they are mostly run as numerical simulations on ordinary workstations. One reason for this are the difficulties determining the weights for the synapses in a network based on analog circuits. The most successful training algorithm is the back-propagation algorithm. It is based on an iteration that calculates correction values from the output error of the network. A prerequisite for this algorithm is the knowledge of the first derivative of the neuron transfer function. While this is easy to accomplish for digital implementations, i.e. ordinary microprocessors and special hardware, it makes analog implementations difficult. The reason for that is that due to device variations, the neurons' transfer functions, and with them their first derivatives, vary from neuron to neuron and from chip to chip. What makes things worse is that they also change with temperature. While it is possible to build analog

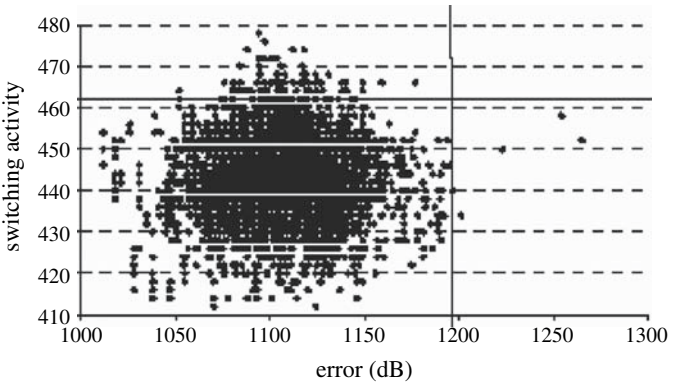


Fig. 7.40 GA search results for 11-bit word length

circuitry that compensates all these effects, this likely results in circuits much larger and slower than their uncompensated counterparts.

To be successful while under a highly competitive pressure from the digital world, analog neural networks should not try to transfer digital concepts to the analog world. Instead they should rely on device physics as much as possible to allow an exploitation of the massive parallelism possible in modern VLSI technologies. Neural networks are well suited for this kind of analog implementation since the compensation of the unavoidable device fluctuations can be incorporated in the weights.

A major problem that still has to be solved is the training. A large number of the analog neural network concepts that can be found in the literature use floating gate technologies like EEPROM of flash memories to store the analog weights. At a first glance this seems to be an optimal solution: it consumes only a small area making therefore very compact synapses possible (down to only one transistor), the analog resolution can be more than 8 bit and the data retention time exceeds 10 years (at 5 bit resolution). The drawback is the programming time and the limited lifetime of the floating gate structure if it is frequently reprogrammed. Therefore such a device needs predetermined weights, but to calculate the weights an exact knowledge of the network transfer function is necessary. To break this vicious circle the weight storage must have a short write time. This would allow a genetic algorithm to come into play. By evaluation of a high number of test configurations the weights could be determined using the real chip. This could also compensate a major part of the device fluctuations, since the fitness data includes the errors caused by these aberrations.

This application describes an analog neural network architecture optimized for genetic algorithms. The synapses are small, $10 \times 10\,\mu m^2$, and fast. The measured network frequency exceeds 50 MHz, resulting in more than 200 giga connections per second for the complete array of 4096 synapses. For building larger networks it should be possible to combine multiple smaller networks, either on the same die or in different chips. This is achieved by confining the analog operation to the synapses and the neuron inputs. The network inputs and neuron outputs are digitally coded. The synapse operation is thereby reduced from a multiplication to an addition. This makes the small synapse size possible and allows the full device mismatch compensation, because each synapse adds either zero or its individual weight that can include any necessary corrections. Analog signals between the different analog network layers are represented by arbitrary multibit connections.

The network presented in this application is optimized for real-time data streams in the range of 1 to 100 MHz and widths of up to 64 bits. This can be used for data transmission applications like high speed DSL1, image processing based on digital edge data produced from camera images by an analog preprocessing chip and for the fitness evaluation of a field programmable transistor array.

7.10.1 Realization of the Neural Network

Principle of operation

Figure 7.41 shows a symbolic representation of a recurrent neural network. Each input neuron (small circle) is linked to each output neuron (large circle) by a synapse (arrow). The output neurons are fed back into the network by a second set of input neurons. The input neurons serve only as amplifiers, while the processing is done at the output neurons.

This architecture allows virtual multilayer networks by choosing the appropriate weights. On the right of Figure 7.41 an example is shown for two layers. Synapse weights set to zero are depicted as dashed arrows. A recurrent network trained by a genetic algorithm has usually no fixed number of layers. Of course, the algorithm can be restricted to a certain number of layers, as in Figure 7.41, but usually it seems to be an advantage to let the genetic algorithm choose the best number of layers. Also, there is no strict boundary between the virtual layers. Each neuron receives input signals from all layers. To avoid wasting synapses if not all the feedback pathways are used, the presented network shares input neurons between external inputs and feedback outputs. Figure 7.42 shows the operation principle of a single neuron. The synaptic weights are stored as charge on a capacitor (storage capacitor).

The neuron operation is separated in two phases, precharge and evaluate. In the precharge phase all the switches in the synapses are set towards the buffer and the precharge signal in the neuron is active. In each synapse the output capacitor is charged via the weight buffer to the same voltage as the storage capacitor. The neuron consists of a comparator and a storage latch. The precharge signal closes a switch between both inputs of the comparator. This precharges the post-synaptic signal to a reference voltage that constitutes the zero level of the network. In the evaluate phase the sum of all the synapses is compared to this precharge voltage. If the synapse signal exceeds it, the neuron fires. This neuron state is stored in the flip-flop at the moment when the phase changes from evaluate to precharge.

In the evaluate phase the synapse switch connects the output capacitor with the post-synaptic signal if the pre-synaptic signal is active. The pre-synaptic signals are

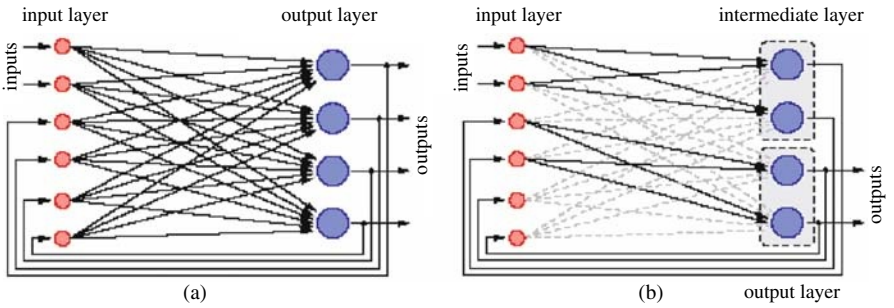


Fig. 7.41 (a): A recurrent neural network. (b): The same network configured as a two layer network

generated by the input neurons depending on the network input and feedback information. This cycle can be repeated a fixed number of times to restrict the network to a maximum layer number and limit the processing time for an input pattern. The network can also run continuously while the input data changes from cycle to cycle. This is useful for signal processing applications.

Figure 7.43 shows a block diagram of the developed neural network prototype. The central element is an array of 64×64 synapses. The post-synaptic lines of the 64 output neurons run horizontally through the synapses, while the presynaptic signals are fed into the array from the input neurons located below and above. Each input neuron can be configured to accept either external data or data from the network itself for input. This internal data comes alternately from two sources. The odd input neurons receive a feedback signal from an output neuron while the even ones get the inverted output from its odd neighbor. If the even neuron is switched to its odd counterpart, they together form a differential input since the binary signal is converted into a pair of alternately active inputs. This is useful for two purposes: if binary coded data is used the number of active input neurons stays always the same, independently of the input data. The second reason is linked to the way the post-synaptic voltage is calculated:

$$V_{postsyn} = \frac{\sum_{i=1}^{64} I_i Q_i}{\sum_{i=1}^{64} I_i C_i} \tag{7.5}$$

Q_i is the charge stored on the synapse output capacitor C_i. I_i is the pre-synaptic signal. As a binary value it is either zero or one. The neuron fires if $V_{postsyn} > V_{precharge}$. Not only the numerator, but also the denominator depends on all the input signals. This has the drawback that if one input signal changes, the other weights' influence on $V_{postsyn}$ changes also. Even though it can be shown that in the simplified model of Eq. 7.5 the network response stays the same, the performance of the real

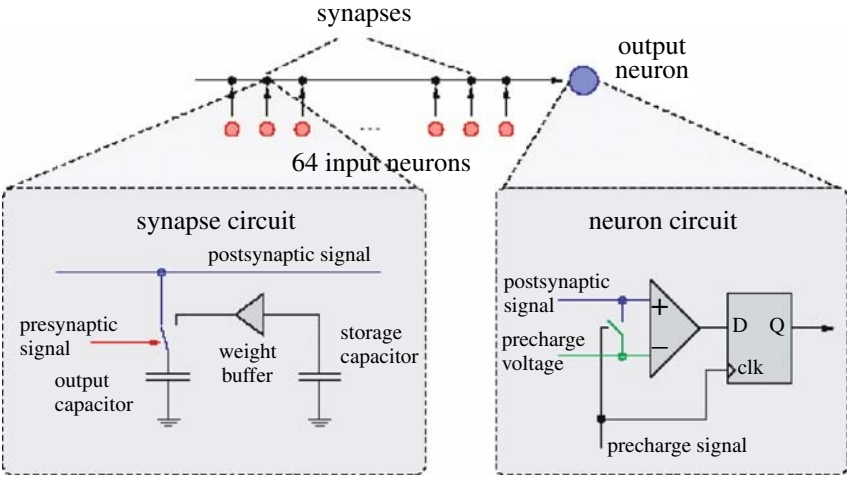


Fig. 7.42 Operation principle of the neuron

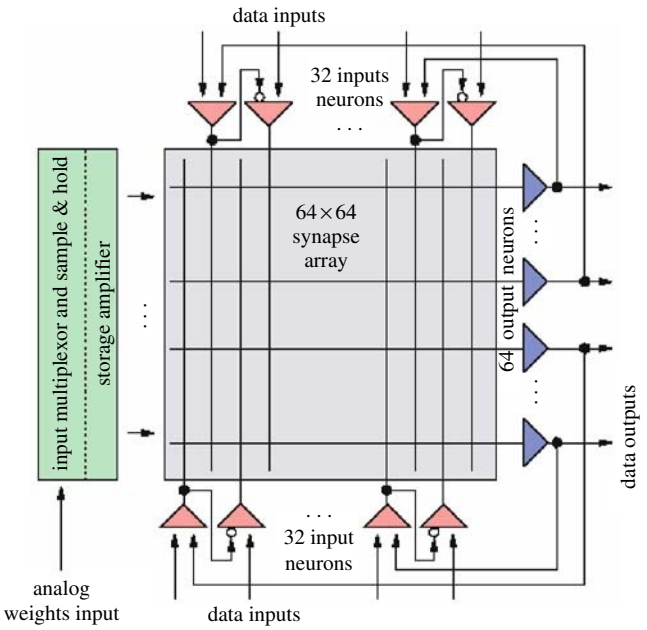


Fig. 7.43 Block diagram of the neural network prototype

network may suffer. The differential input mode avoids this effect by activating always one input neuron per data input. The capacitance switched onto the post-synaptic signal line becomes independent of the data. Therefore the denominator of Eq. 7.5 stays the same for any changes of a differential input. The disadvantage is the reduced number of independent inputs since each differential input combines an odd with an even input neuron.

Implementation of the Network Circuits

A micro photograph of the fabricated chip can be seen in Figure 7.44. The technology used is a 0.35 µm CMOS process with one poly and three metal layers. The die size is determined by the IO pads necessary to communicate with the test system. The synapse array itself occupies less than 0.5 mm^2. It operates from a single 3.3 volt supply and consumes about 50 mW of electrical power. Figure 7.45 shows the circuit diagram of the synapse circuit. Both capacitors are implemented with MOS-transistors. The weight buffer is realized as a source follower built from the devices M1 and M2. The offset and the gain of this source follower vary with the bias voltage as well as the temperature. Therefore an operational amplifier outside of the synapse array corrects the weight input voltage until the output voltage of the source follower equals the desired weight voltage which is fed back via M7. The charge injection error caused by M6 depends on the factory induced mismatch that

can be compensated by the weight value. M3 is closed in the precharge phase of the network to charge the output capacitor to the weight voltage.

M5 speeds up this process by fully charging the capacitor first. Since the output current of the source follower is much larger for a current flowing out of M1 instead of into M2 it discharges the capacitor faster than it is able to charge it. The total time to charge the output capacitor to the desired voltage decreases therefore by this combination of discharging and charging. In the evaluate phase M4 is enabled by the pre-synaptic signal of the input neuron connected to the synapse. Charge sharing between all the enabled synapses of every output neuron takes place on the post-synaptic lines. In Figure 7.46 a part of the layout drawing of the synapse array is shown. Most of the area is used up by the two capacitances. The values for the storage and output capacitances are about 60 and 100 fF respectively. The charge on the storage capacitors must be periodically refreshed due to the transistor leakage currents. In the training phase this happens automatically when

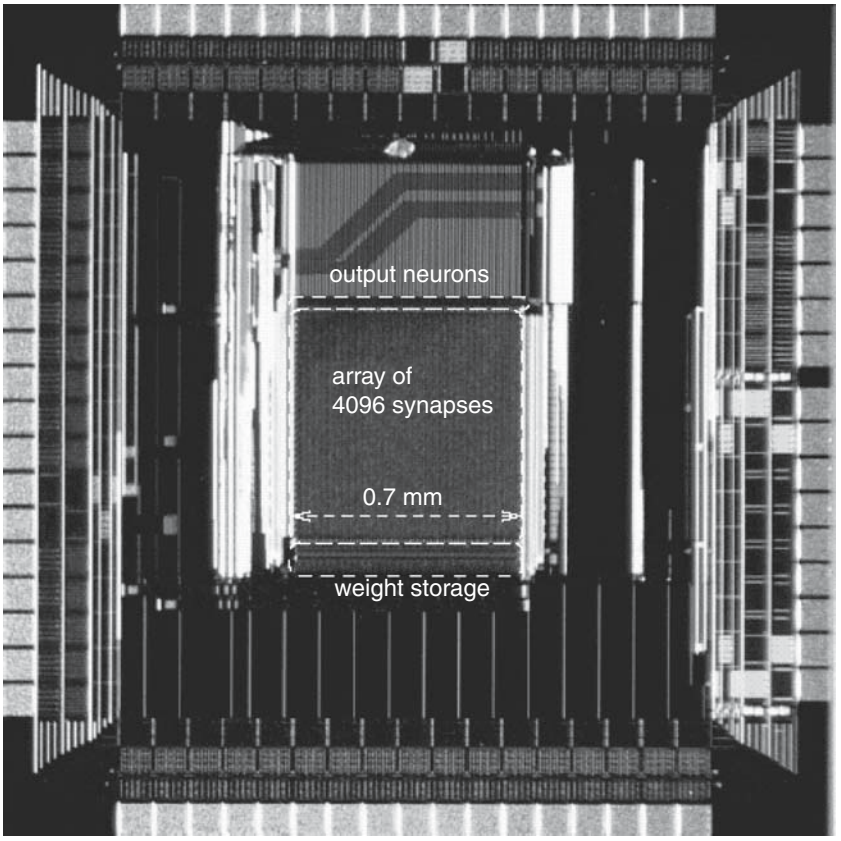


Fig. 7.44 Micro photograph of the neural network chip

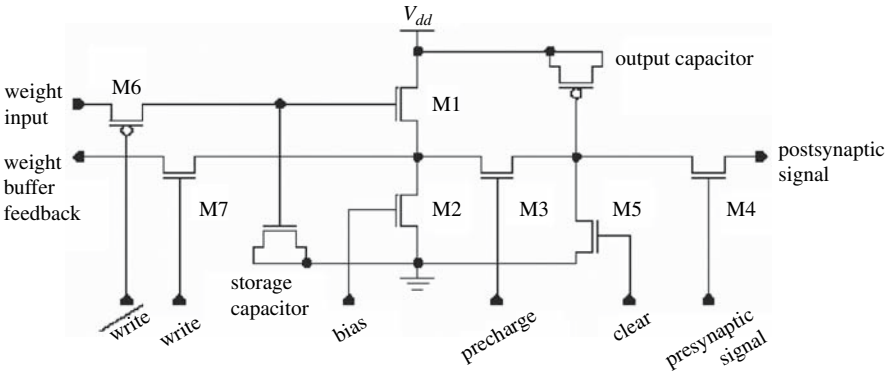


Fig. 7.45 Circuit diagram of the synapse

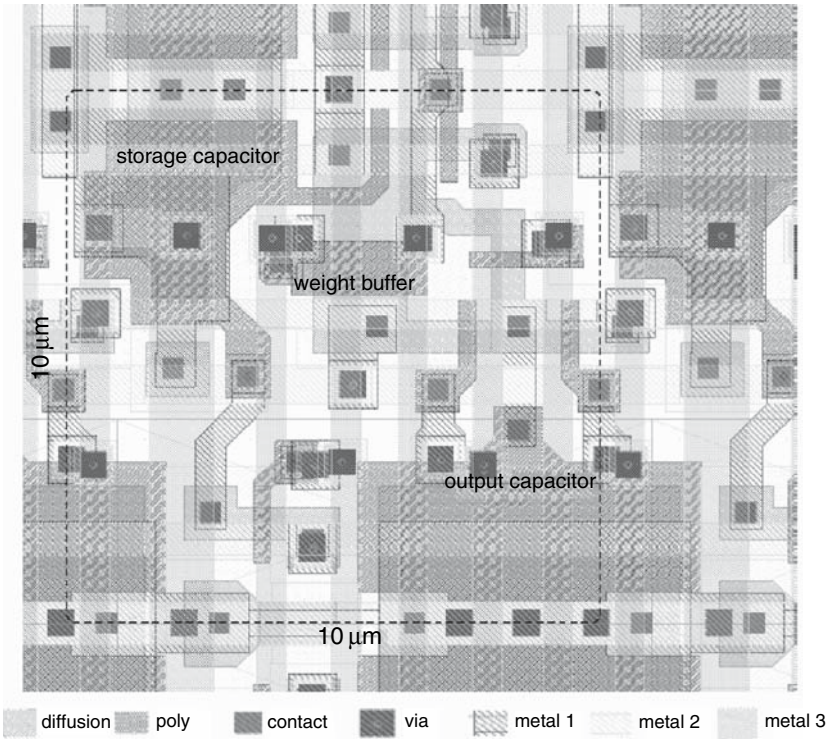


Fig. 7.46 Layout drawing of the synapse array showing one synapse

the weights are updated, otherwise the refresh takes up about 2 % of the network capacity.

Figure 7.47 shows the circuit diagram of the neuron circuit. It is based on a sense amplifier built from the devices M1 to M4. In the precharge phase it is disabled

(evaluate and evaluate are set to the precharge voltage). The activated precharge and transfer signals restore the post-synaptic input signal and the internal nodes of the sense amplifier to the precharge voltage. At the beginning of the evaluate phase the precharge signal is deactivated while transfer stays on. The potential on the post-synaptic input changes now by the activated synapses. Transistor M5 transfers it onto the gates of M3 and M4. The small differential voltage between the gates of M1/M2 and M3/M4 is amplified by disabling transfer and activating evaluate/evaluate. At the same moment the synapses switch back to the precharge phase. The sense amplifier restores the signal in about 1 ns to the full supply voltage. With the read signal the result is stored in the output latch formed by the inverters I1 and I2. The output of I1 is fed back to the input neurons. The output neuron forms a master/slave flip-flop with the sense amplifier as the master and the output latch as the slave. This results in a discrete-time operation of the network. Together with the feedback the network acts as a kind of mixed-signal state machine. The neurons are the state flip-flops while the synapse array represents the logic that determines the next state. The simulated maximum clock frequency of the network is 100 MHz.

7.10.2 Implementation of the Genetic Training Algorithm

The time needed to load the 4096 weight values into the network is about 250 µs. A single test pattern comprised of 64 input bits can be applied in about 100 ns. This makes it feasible to use iterative algorithms needing high numbers of passes. The dependency between a weight value and a neuron output could be highly nonlinear, especially if more than one network cycle is used to implement a multilayered recurrent network. Therefore a genetic algorithm seems to be well suited to train the network. The network has also built-in hardware support for perturbation based learning, an iterative algorithm that needs no knowledge about the transfer function of the network. The implemented genetic algorithm represents one weight value by one gene. To avoid close hardware dependencies the weight

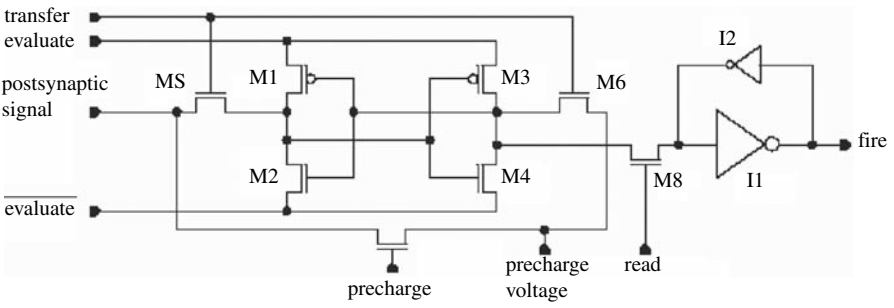


Fig. 7.47 Circuit diagram of the neuron

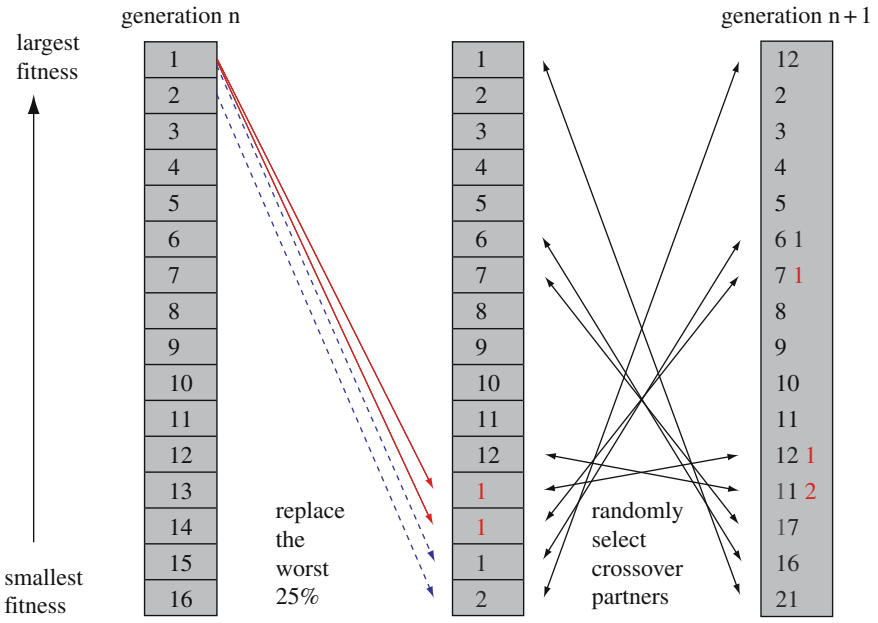


Fig. 7.48 Crossover pattern used in the genetic algorithm

value is stored in a normalized way using floating point numbers between -1 for the maximum inhibitory and $+1$ for the maximum excitatory synapse. These numbers are converted into the voltage values needed by the analog neural network while translating the genome into the weight matrix. The genes comprising one neuron are combined to a chromosome. Up to 64 chromosomes form the genome of one individual. The artificial evolution is always started by creating a random population. After an individual has been loaded into the weight matrix the test patterns are applied. The fitness is calculated by comparing the output of the network with the target values. For each correct bit the fitness is increased by one. This is repeated for the whole population. After sorting the population by the fitness two genetic operators are applied: crossover and mutation. The crossover strategy is depicted in Figure 7.48. It shows an example for a population of 16 individuals.

The worst 25% of the population are replaced in equal halves by the fittest individual (solid arrows) and the 12.5 % best ones (dashed arrows). 75% of the individuals are kept unchanged. As shown in Figure 7.48 the crossover partners are randomly chosen. The crossover itself is done in a way that for each pair of identical chromosomes (i.e. chromosomes coding the same output neuron) a crossover point is randomly selected. All the genes up to this point are exchanged between both chromosomes. After the crossover is done the mutation operator is applied on the

new population. It alters every gene with equal probability. If a gene is subject to mutation, its old value is replaced by a randomly selected new one (again out of the range $[-1;\ 1]$).

7.10.3 Experimental Results

The network and the genetic training algorithm have been tested with the setup shown in Figure 7.49. The population is maintained on the host computer.

The data for each individual is sent via the FPGA to the neural network using a 16 bit digital to analog converter to generate the analog weight values from the gene data. The testpatterns and the target data are stored in the RAM on the test-board throughout the evolutionary process. They are applied to the individual after the neural network has stored its weights. The FPGA reads the results and calculates the fitness. After the last testpattern the final fitness value is read back by the host computer and the test of the next individual starts. To speed up this process the weight data for the next individual can be uploaded into the RAM while the testpatterns are applied to the current individual. Since the test board is not yet capable of the full speed of the network the number of individuals tested per second is limited to about 150 to 300, depending on the number of testpatterns used. To test the capability of the genetic algorithm a training pattern was chosen that is especially hard to learn with traditional algorithms like back-propagation: the calculation of parity. While easy to implement with exclusive-or gates, it can not be learned by a single layered neural network. Therefore it also shows the ability of the presented neural network to act as a two-layered network. Figure 7.50 shows the testpattern definition for an eight bit parity calculation. Since the input data is binary coded, the input neurons are configured for differential input. The number of network cycles is set to two and four output neurons are fed back into the network. This allows the genetic algorithm to use an internal layer with four neurons. For each testpattern one target bit is defined: the parity of the testpattern. Figures 7.51 and 7.52 show plots of the fitness versus the generation number for different parity experiments.

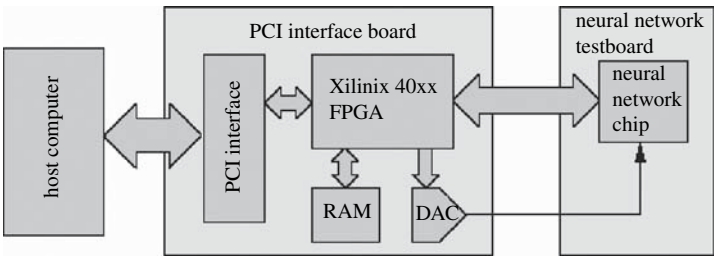


Fig. 7.49 Testbench used for the evaluation of the neural network

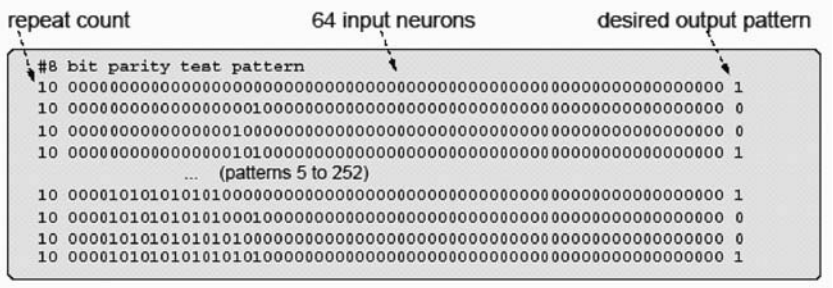


Fig. 7.50 Testpattern definition for the 8 bit parity

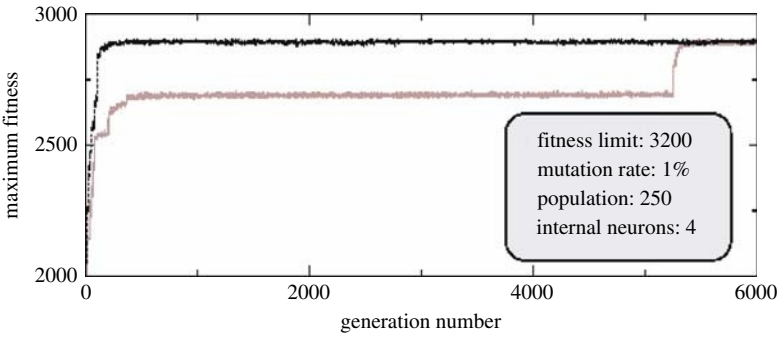


Fig. 7.51 Example fitness curves from 6 bit parity experiments

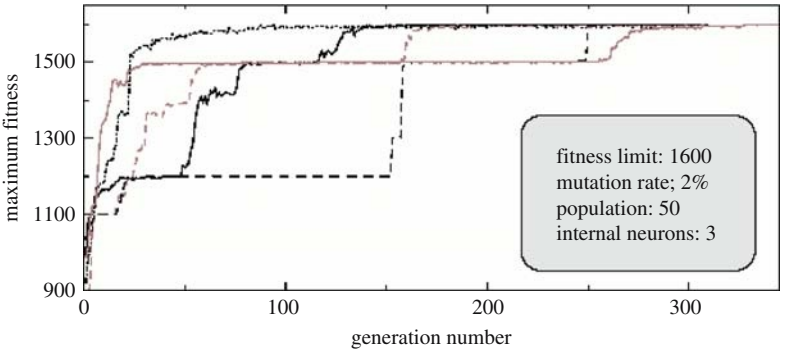


Fig. 7.52 Example fitness curves from 4 bit parity experiments

At 6 bit the network does not learn all the patterns any more. The random nature of the artificial evolution is clearly visible: the black curve approaches the same fitness as the gray one about 5000 generations earlier.

7.10.4 Concluding Remarks

This application presents a new architecture for analog neural networks that is optimized for iterative training algorithms, especially genetic algorithms. By combining digital information exchange with analog neuron operation it is well suited for large neural network chips. Especially, the very small synapse area makes network chips with more than a million synapses possible. The mapping of input data to the network and the effective number and size of the network layers is programmable. Therefore not only the weights, but also a significant part of the architecture can be evolved. The accuracy of the network is not limited by the precision of a single analog synapse since arbitrary synapses can be combined.

By implementing this task in the genetic algorithm, the network could automatically adapt its prediction performance to a specific data set. The presented prototype successfully learned the parity calculation of multibit patterns. This shows that genetic algorithms are capable of training two layered analog neural networks. At the time of this writing the test setup was limited in its analog precision. This makes it difficult to train binary patterns of 6 or more bits without errors. Also, the genetic algorithm used is a first approach to show the functionality of the system. These limitations will be hopefully overcome in the near future.

Chapter 8
Genetic Programming Applications

Learning Objectives: On completion of this chapter the reader will have knowledge on applications of Genetic Algorithms such as:

- Gp-robocode: using genetic programming to evolve robocode players
- Prediction of biochemical reactions using genetic programming
- Application of genetic programming to high energy physics event selection
- Using genetic programming to generate protocol adaptors for interprocess communication
- Improving technical analysis predictions: an application of genetic programming
- Genetic programming within civil engineering
- Chemical process controller design using genetic programming
- Trading applications of genetic programming
- Artificial neural network development by means of genetic programming with graph codification

8.1 GP-Robocode: Using Genetic Programming to Evolve Robocode Players

The strife between humans and machines in the arena of intelligence has fertilized the imagination of many an artificial-intelligence (AI) researchers, as well as numerous science fiction novelists. Since the very early days of AI, the domain of games has been considered as epitomic and prototypical of this struggle. Designing a machine capable of defeating human players is a prime goal in this area: From board games, such as chess and checkers, through card games, to computer adventure games and 3D shooters, AI plays a central role in the attempt to see machine beat man at his own game literally. Program-based games are a subset of the domain of games in which the human player has no direct influence on the course of the game; rather, the actions during the game are controlled by programs that were written by the (usually human) programmer. The program responds to the

current game environment, as captured by its percepts, in order to act within the simulated game world. The winner of such a game is the programmer who has provided the best program; hence, the programming of game strategies is often used to measure the performance of AI algorithms and methodologies. Some famous examples of program-based games are RoboCup, the robotic soccer world championship, and CoreWars, in which assembly-like programs struggle for limited computer resources. While the majority of the programmers actually write the code for their players, some of them choose to use machine-learning methods instead. These methods involve a process of constant code modifications, according to the nature of the problem, in order to achieve as best a program as possible. If the traditional programming methods focus on the ways to solve the problem (the 'how'), machine-learning methods focus on the problem itself (the 'what') to evaluate the program and constantly improve the solution. This application has chosen the game of Robocode, a simulation-based game in which robotic tanks fight to destruction in a closed arena. The programmers implement their robots in the Java programming language, and can test their creations either by using a graphical environment in which battles are held, or by submitting them to a central web site where online tournaments regularly take place; this latter enables the assignment of a relative ranking by an absolute yardstick, as is done, e.g., by the Chess Federation. The game has attracted hundreds of human programmers and their submitted strategies show much originality, diversity, and ingenuity. One of the major objectives is to attain what Koza and his colleagues have recently termed human-competitive machine intelligence. The result holds its own or wins a regulated competition involving human contestants (in the form of either live human players or human-written computer programs). Since the vast majority of Robocode strategies submitted to the league were coded by hand, this game is ideally suited to attain the goal of human competitiveness. The machine-learning method chosen here is Genetic Programming (GP), in which the code for the player is created through evolution. The code produced by GP consists of a tree-like structure, which is highly exible, as opposed to other machine-learning techniques (e.g., neural networks).

8.1.1 Robocode Rules

A Robocode player is written as an event-driven Java program. A main loop controls the tank activities, which can be interrupted on various occasions, called events. Whenever an event takes place, a special code segment is activated, according to the given event. For example, when a tank bumps into a wall, the HitWallEvent will be handled, activating a function named onHitWall(). Other events include: hitting another tank, spotting another tank, and getting hit by an enemy shell. There are five actuators controlling the tank: movement actuator (forward and backward), tank-body rotation actuator, gun-rotation actuator, radar rotation actuator, and fire actuator (which acts as both trigger and firepower controller). As the round begins,

each tank of the several placed in the arena is assigned a fixed value of energy. When the energy meter drops to zero, the tank is disabled, and if it hits, it is immediately destroyed. During the course of the match, energy levels may increase or decrease: a tank gains energy by firing shells and hitting other tanks, and loses energy by getting hit by shells, other tanks, or walls. Firing shells costs energy. The energy lost when firing a shell, or gained, in case of a successful hit, is proportional to the firepower of the fired shell. The round ends when only one tank remains in the battle field (or no tanks at all), whereupon the participants are assigned scores that reflect their performance during the round. A battle lasts a fixed number of rounds. In order to test the evolved Robocode players and compare them to human written strategies they were submitted to the international league. The league comprises a number of divisions, classified mainly according to allowed code size. When the players play in duels, and their code is limited to four instances of a semicolon (four lines), with no further restriction on code size. Since GP naturally produces long lines of code, this league seemed most appropriate for the research. GP produces longer programs due to much junk DNA.

8.1.2 Evolving Robocode Strategies using Genetic Programming

Here Koza-style GP is used, in which a population of individuals evolves. An individual is represented by an ensemble of LISP expressions, each composed of functions and terminals. The functions used are mainly arithmetic and logical ones, which receive several arguments and produce a numeric result. Terminals are zero-argument functions, which produce a numerical value without receiving any input. The terminal set is composed of zero-argument mathematical functions, robot perceptual data, and numeric constants. The list of functions and terminals is given in Table 8.1, and will be described below. In this application Strongly Typed Genetic Programming (STGP) is used, in which functions and terminals differ in types and are restricted to the use of specific types of inputs; another technique was the use of Automatically Define Functions (ADFs), which enables the evolution of subroutines. These techniques and a number of others proved not to be useful for the game of Robocode. GP is used to evolve numerical expressions that will be given as arguments to the player's actuators. As mentioned above, the players consist of only four lines of code (each ending with a semicolon). However, there is much variability in the layout of the code: decision has to be made which events are to be implemented, and which actuators would be used for these events.

To obtain the strict code-line limit, the following adjustments were made:

- Omit the radar rotation command. The radar, mounted on the gun, was instructed to turn using the gun-rotation actuator.
- Implement the fire actuator as a numerical constant which can appear at any point within the evolved code sequence (Table 8.1).

Table 8.1 GP Robocode system: functions and terminals

Game Status Indicators	
Energy()	Returns the remaining energy of the player
Heading()	Returns the current heading of the player
X()	Returns the current horizontal position of the player
Y()	Returns the current vertical position of the player
MaxX()	Returns the horizontal battlefield dimension
MaxY()	Returns the vertical battlefield dimension
EnemyBearing()	Returns the current enemy bearing, relative to the current player's heading
EnemyDistance()	Returns the current distance to the enemy
EnemyVelocity()	Returns the current enemy's velocity
EnemyHeading()	Returns the current enemy heading, relative to the current player's heading
EnemyEnergy()	Returns the remaining energy of the enemy
Numerical constants	
Constant()	An ERC in the range $[-1, 1]$
Random()	Returns a random real number in the range $[-1, 1]$
Zero()	Returns the constant 0
Arithmetic and logical functions	
Add(x; y)	Adds x and y
Sub(x; y)	Subtracts y from x
Mul(x; y)	Multiplies x by y
Div(x; y)	Divides x by y, if y is nonzero; otherwise returns 0
Abs(x)	Returns the absolute value of x
Neg(x)	Returns the negative value of x
Sin(x)	Returns the function sin(x)
Cos(x)	Returns the function cos(x)
ArcSin(x)	Returns the function arcsin(x)
ArcCos(x)	Returns the function arccos(x)
IfGreater(x; y; exp1; exp2)	If x is greater than y returns the expression exp1, otherwise returns the expression exp2
IfPositive(x; exp1; exp2)	If x is positive, returns the expression exp1, otherwise returns the expression exp2
Fire command	
Fire(x)	If x is positive, executes a fire command with x being the firepower, and returns 1; otherwise, does nothing and returns 0

The main loop contains one line of code that directs the robot to start turning the gun (and the mounted radar) to the right. This insures that within the first gun cycle, an enemy tank will be spotted by the radar, triggering a Scanned Robot Event. Within the code for this event, three additional lines of code were added, each controlling a single actuator, and using a single numerical input that was evolved using GP. The first line instructs the tank to move to a distance specified by the first evolved argument. The second line instructs the tank to turn to an azimuth specified by the second evolved argument. The third line instructs the gun (and radar) to turn to an azimuth specified by the third evolved argument (Figure 8.1).

Robocode Player's Code Layout

```
while (true)
TurnGunRight(INFINITY); //main code loop
...
OnScannedRobot() {
MoveTank(< GP#1 >);
TurnTankRight(< GP#2 >);
TurnGunRight(< GP#3 >);
}
```

Fig. 8.1 Robocode player's code layout

Functions and Terminals

Since terminals are actually zero-argument functions, it was found that the difference between functions and terminals was of little importance. Instead, the terminals and functions were divided into four groups according to their functionality:

1. Game-status indicators: A set of terminals that provide real-time information on the game status, such as last enemy azimuth, current tank position, and energy levels.
2. Numerical constants: Two terminals, one providing the constant 0, the other being an ERC (Ephemeral Random Constant), as described by Koza. This latter terminal is initialized to a random real numerical value in the range $[-1, 1]$, and does not change during evolution.
3. Arithmetic and logical functions: A set of zero- to four-argument functions, as specified in Table 8.1.
4. Fire command: This special function is used to preserve one line of code by not implementing the fire actuator in a dedicated line. The exact functionality of this function is described in Table 8.1.

Fitness Measure

The fitness measure should reflect the individual's quality according to the problem at hand. When choosing a fitness measure for the Robocode players, two main considerations were in mind: the opponents and the calculation of the fitness value itself. Selection of opponents and number of battle rounds: A good Robocode player should be able to beat many different adversaries. Since the players in the online league differ in behavior, it is generally unwise to assign a fitness value according to a single-adversary match. On the other hand, it is unrealistic to do battle with the entire player set not only is this a time-consuming task, but new adversaries enter the tournaments regularly. Several opponent set sizes were tested, including from one to five adversaries. Some of the tested evolutionary configurations involved random selection of adversaries per individual or per generation, while other configurations

consisted of a fixed group of adversaries. The configuration ultimately chosen was to use involved a set of three adversaries fixed throughout the evolutionary run, with unique behavioral patterns. Since the game is nondeterministic, a total of three rounds were played versus each adversary to reduce the randomness factor of the results.

Calculation of the Fitness Value

Since fitness is crucial in determining the trajectory of the evolutionary process, it is essential to find a way to translate battle results into an appropriate fitness value. The goal was to excel in the online tournaments; hence, the scoring algorithms were adopted in these leagues. The basic scoring measure is the fractional score F, which is computed using the score gained by the player S_P and the score gained by its adversary S_A:

$$F = \frac{S_P}{S_P + S_A}$$

This method reflects the player's skills in relation to its opponent. It encourages the player not only to maximize its own score, but to do so at the expense of its adversary's. It is observed that in early stages of evolution, most players attained a fitness of zero, because they could not gain a single point in the course of the battle. To boost population variance at early stages, a modified fitness measure $\overline{F}$ was devised:

$$\overline{F} = \frac{\in + S_P}{\in + S_P + S_A}$$

where $\in$ is a fixed small real constant.

This measure is similar to the fractional-score measure, with one exception: when two evolved players obtain no points at all (most common during the first few generations), a higher fitness value will be assigned to the one which avoided its adversary best (i.e., lower S_A). This proved sufficient in enhancing population diversity during the initial phase of evolution. When facing multiple adversaries, the average modified fractional score was used, over the battles against each adversary.

Evolutionary Parameters

1. Population size: 256 individuals.
2. Termination criterion and generation count: No limit for the generation count in the evolutionary runs. Instead, the run was simply stopped manually when the fitness value stopped improving for several generations.

3. Creation of initial population: Koza's ramped-half-and-half method was used, in which a number d, between mindepth (set to 4) and maxdepth (set to 6) is chosen randomly for each individual. The genome trees of half of the individuals are then grown using the Grow method, which generates trees of any depth between 1 and d, and the other half is grown using the Full method, which generates trees of depth d exactly. All trees are generated randomly, by selection of appropriate functions and terminals in accordance with the growth method.
4. Breeding operators: Creating a new generation from the current one involves the application of genetic operators (namely, crossover and mutation) on the individuals of the extant generation. Here two such operators were used:

   a. Mutation (unary): randomly selects one tree node (with probability 0.9) or leaf (with probability 0.1), deletes the subtree rooted at that node and grows a new subtree instead, using the Grow method. Bloat control is achieved by setting a maxdepth parameter (set to 10), and invoking the growth method with this limit.
   b. Crossover (binary): randomly selects a node (with probability 0.9) or a leaf (with probability 0.1) from each tree, and switches the subtrees rooted at these nodes. Bloat control is achieved using Langdon's method, which ensures that the resulting trees do not exceed the maxdepth parameter (set to 10).

      The breeding process starts with a random selection of genetic operator: a probability of 0.95 of selecting the crossover operator, and 0.05 of selecting the mutation operator. Then, a selection of individuals is performed (as described in the next paragraph): one individual for mutation, or two for crossover. The resulting individuals are then passed on to the next generation.

5. Selection method: Tournament selection was used, in which a group of individuals of size k (set to 5) is randomly chosen. The individual with the highest fitness value is then selected. The two highest fitness individuals were passed to the next generation with no modifications.
6. Extraction of best individual: When an evolutionary run ends, the best evolved individuals should be determined. Since the game is highly nondeterministic, the fitness measure does not explicitly reflect the quality of the individual: a lucky individual might attain a higher fitness value than better overall individuals. In order to obtain a more accurate measure for the players evolved in the last generation, they were allowed to battle for 100 rounds against 12 different adversaries (one at a time). The results were used to extract the optimal player to be submitted to the league.

Genetic programming is known to be time consuming, mainly due to fitness calculation. The time required for one run can be estimated using this simple equation:

$$ExceutionTime = RoundTime * NumRounds * NumAdversaries * PopulationSize * NumGenerations$$

A typical run involved 256 individuals, each battle carried out for 3 rounds against 3 different adversaries. A single round lasted about one second, and the best evolutionary run took approximately 400 generations, so the resulting total run time was:

Execution Time $= 1 \times 3 \times 3 \times 256 \times 400 \approx 9.2 \times 10^5$ seconds $=$ 256 hours;
or about 10 days.

8.1.3 Results

Multiple evolutionary runs were performed against three leading opponents. The progression of the best run is shown in Figure 8.2.

Due to the nondeterministic nature of the Robocode game, and the relatively small number of rounds played by each individual, the average fitness is worthy of attention, in addition to the best fitness. The first observation to be made is that the average fractional score converged to a value equaling 0:5, meaning that the average Robocode player was able to hold its own against its adversaries. When examining the average fitness, one should consider the variance: A player might defeat one opponent with relatively high score, while losing to the two others.

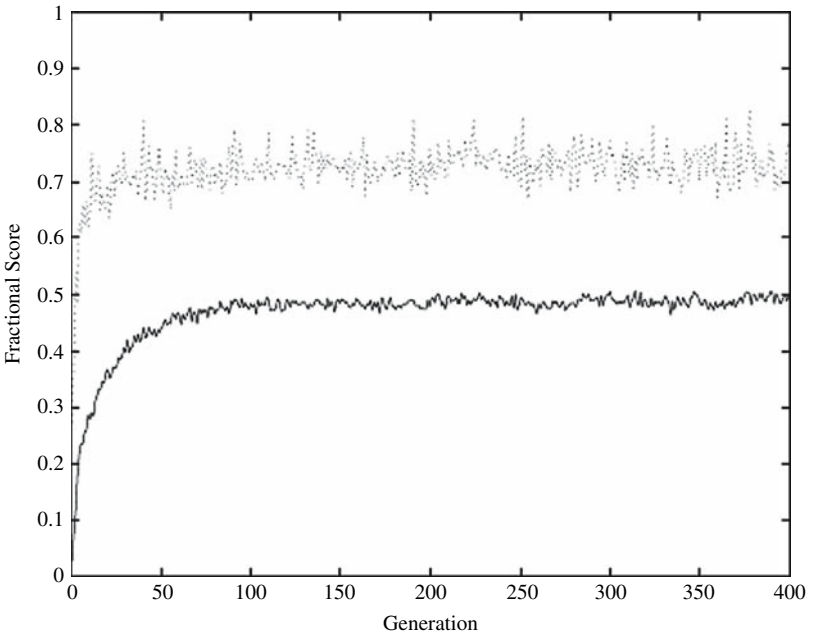


Fig. 8.2 Modified fractional score averaged over three different adversaries, versus time (generations). Top (dotted) curve: best individual, bottom (solid) curve: population average

Though an average fitness of 0.5 might not seem impressive, two comments are in order:

1. This value reflects the average fitness of the population; some individuals attain much higher fitness.
2. The adversaries used for fitness evaluation were excellent ones, taken from the top of the HaikuBot league. In the real world, the evolved players faced a greater number of adversaries, most of them inferior to those used in the evolutionary process.

8.1.4 Concluding Remarks

The result holds its own or wins a regulated competition involving human contestants (in the form of either live human players or human-written computer programs). The game of Robocode, being nondeterministic, continuous, and highly diverse (due to the unique nature of each contestant), induces a virtually infinite search space, making it an extremely complex (and thus interesting) challenge for the GP. When performing an evolutionary run against a single adversary, winning strategies were always evolved. However, these strategies were specialized for the given adversary: When playing against other opponents (even relatively inferior ones), the evolved players were usually beaten. Trying to avoid this obstacle, evolutionary runs included multiple adversaries, resulting in better generalization, as evidenced by the league results.

8.2 Prediction of Biochemical Reactions using Genetic Programming

To comprehend dynamic behaviors of biological systems, many models have been proposed. These models need literatures data to represent detailed and accurate dynamics. However, those data are sufficient only in few cases. To solve this problem, many techniques have been developed, including Genetic Algorithm (GA). Those methods require defining equations before predicting biological systems. To consider the case where even equation could not be obtained, the Genetic Programming (GP) is employed as a method to predict arbitrary equation from time course data without any knowledge of the equation. However, it is difficult for conventional GP to search the equations with high accuracy because target biochemical reactions include not only variables but also many numerical parameters. In order to improve the accuracy of GP, elite strategy is extended to focus on numerical parameters. In addition, a penalty term was added to evaluation function to save the growth of the size of tree and consuming calculation time. The relative square error of predicted and given time-course data were decreased from 25.4% to 0.744%. Moreover, in

experiments to validate the generalization ability of the predicted equations, the relative square error was successfully decreased of the predicted and given time-course data from 25.7% to 0.836%. The results indicate that the method succeeded in predicting approximation formulas without any definition of equations with reduced square error.

8.2.1 Method and Results

To validate the proposed method, some numerical experiments were conducted. As a case study, the equation about pure competitive inhibition reaction was tried by 2 different exclusive inhibitors. The numerical parameters of the above equation was artificially prepared as below.

$$\frac{d[P]}{dt} = \frac{1.000[S]}{4.800 + [S] + 1.120[I] + 0.570[X]}$$

where [S] and [P] are the concentrations of substances.

The multiple time-course data sets with different initial concentration are generated by the equation and was predicted using the proposed method from those time-course data sets. One of the results of predicted equations is shown as below.

$$\frac{d[P]}{dt} = \frac{[S]}{5.631 + [S] + [I] + 0.450[X]}$$

Time-course data and predicted equations were prepared from those time-courses. One of the results of given and simulated time-course is shown in Figure 8.3. Transition of evaluated value of conventional method and proposed one is shown in Figure 8.4.

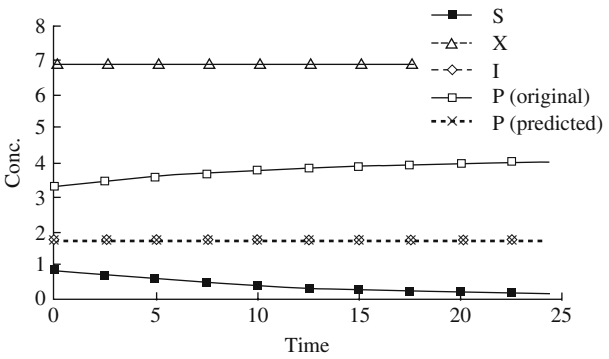


Fig. 8.3 One of the results of given time- course data and time-course data simulated by predicted equation. Dos denote sampling points. These points were selected artificially as a case study. The square error of concentration of substance P of time-course data and time-course data simulated by predicted equation is 0.147%

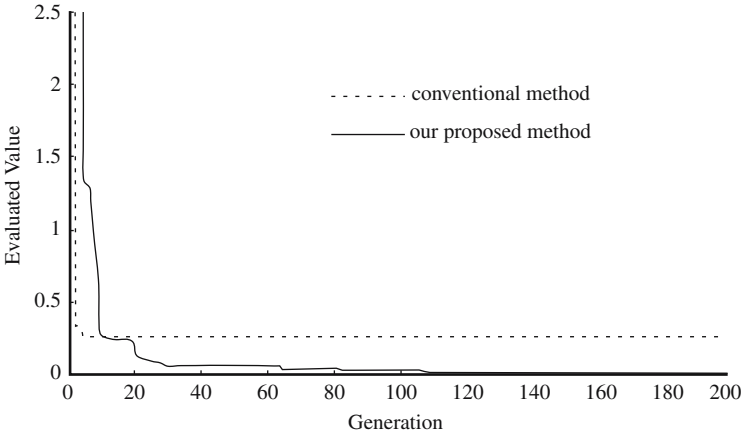


Fig. 8.4 Transition of evaluated value of conventional method and proposed method. The dotted curve is the evaluated value calculated by conventional method. The solid one is that calculated by proposed method. The average of relative square error by conventional method is 25.4% and that by the proposed method is 0.744%

8.2.2 Discussion

The proposed method could find the equation whose dimension of variables are strictly same as originals and improve the accuracy compared to conventional GP. However, the numerical parameters of the denominator of predicted equations were a little different from the original so that the predicted equation was not able to calculate the initial dynamics satisfactorily. A simple means of avoiding such problems is to increase the number of sampling points. However, it is important to avoid over fitting that worsens the ability to find the equation. The number of sampling points is a trade-off between those problems. In future investigations it is proposed to develop a method to find the appropriate number of sampling points automatically.

8.3 Application of Genetic Programming to High Energy Physics Event Selection

Genetic programming is one of a number of machine learning techniques in which a computer program is given the elements of possible solutions to the problem. This program, through a feedback mechanism, attempts to discover the best solution to the problem at hand, based on the programmers definition of success. Genetic programming differs from machine learning solutions such as genetic algorithms and neural networks in that the form or scope of the solution is not specified in advance, but is determined by the complexity of the problem. This technique is

applied to the study of a doubly Cabibbo suppressed branching ratio of D+ since this measurement is presumed to be reasonably free of Monte Carlo related systematic errors. This is the first application of the genetic programming technique to high energy physics data.

8.3.1 Genetic Programming

Initial Tree Generation

The initial trees to be evaluated are created in one of two ways. The first, termed "grow," randomly selects either a terminal or a function as the first (top) node in the tree. If a function is selected, the "child" nodes required by that function are again randomly selected from either functions or terminals. Growth is stopped either when all available nodes have been filled with terminals or when the maximum depth (measured from top to bottom) for the tree is reached (at which point only terminals are chosen). The second method, termed "full," proceeds similarly, except only functions are chosen at levels less than the desired depth of the initial tree and only terminals are selected at the desired depth. Typically one specifies a range of depths, which results in subpopulations of differing complexity. In the initial generation, every tree in the population is guaranteed to be unique. Later generations are allowed to have duplicate individuals. This is not quite accurate. Every tree within a sub-process is unique; it runs with up to 40 sub-processes, as explained later. For a wide range of problems it has been observed that generating half the initial population with the grow method and the other half with various depths of full trees provides a good balance in generating the initial population.

Fitness Evaluation

Central to genetic programming is the idea of fitness. Fitness is the measure of how well the program, or tree, solves the problem presented by the user. The GP requires the calculation of *standardized fitness* such that the best possible solution will have a fitness of 0 and all other solutions will have positive fitness values. The exact definition of the standardized fitness is left to the programmer. Another useful quantity is the adjusted fitness,

$$f_a(i) = \frac{1}{1 + f_s(i)}$$

where $f_a(i)$ is the adjusted fitness of the ith individual and $f_s(i)$ is the standardized fitness of the same individual. It is observed that as f_s decreases, f_a increases to a maximum of 1.0.

Survival Selection

To mimic the evolutionary process of natural selection, the probability that a particular tree will pass some of its "genetic material," or instructions, on to the next generation must be proportional in some way to the fitness of that individual. In genetic programming, several kinds of selection are employed. The first is "fitness-proportionate," in which the probability, p_i, of selection of the ith individual is

$$p_i = \frac{f_a(i)}{\sum_j f_a(j)}$$

where j sums over all the individuals in a population. In this way, very fit individuals are selected more often than relatively unfit individuals. This is the most common method of selection in genetic programming. For complicated problems which require larger population sizes, "fitness-overselection" is often used. In this method, the population is divided into two groups, a small group of "good" individuals and a larger group of the remaining individuals. The majority of the time, an individual is selected from the smaller group, using the rules for fitness-proportionate selection. The rest of the time, the individual is selected from the larger group, again with the same rules. In the implementation, the small group contains the best individuals which account for (320/population size) of the total adjusted fitness. E.g., for a population size of 4000, the small group contains the individuals summing to % of the total adjusted fitness. Individuals are selected from this small group 0% of the time. Additional types of selection are sometimes used in genetic programming. These include "tournament," in which two or more individuals are randomly chosen and the best is selected, and "best," in which the best individuals are elected in order.

Breeding Algorithms

The process of creating a new generation of programs from the preceding generation is called "breeding" in the genetic programming literature. The similar sounding term, "reproduction," is used in a specific context as explained below. The methods used to produce new individuals are determined randomly; more than one method can be (and is) used. Each of these methods must select one or more individuals as "parents." The parents are selected according to the methods described above. The GP used implements three such methods: reproduction, crossover, and mutation. Other methods, e.g. permutation, also exist.

Migration

Although not part of early genetic programming models, migration of individuals is used as an important element of generating diversity. In *parallel* genetic programming model, sub-populations (or "islands") are allowed to evolve independently

with exchanges between sub-populations taking place every few generations. Every ng generations, the best ni individuals from each sub-population are copied into every other sub-population. After this copying, they may be selected by the methods discussed above for the various breeding processes. This modification to the method allows for very large effective population to be spread over a large number of computers in a network.

Termination

When the required number of generations has been run and all the individuals evaluated, the GP terminates. At this point, a text representation of the best tree (not necessarily from the last generation) is written out and other cleanup tasks are performed.

8.3.2 Combining Genetic Programming with High Energy Physics Data

While genetic programming techniques can be applied to a wide variety of problems in High Energy Physics (HEP), the technique is illustrated on the common problem of event selection: the process of distinguishing signal events from the more copious background processes. Event selection in HEP has typically been performed by the "cut method." The physicist constructs a number of interesting variables, either representing kinematic observables or a statistical match between data and predictions. These variables are then individually required to be greater and/or less than some specified value. This increases the purity of the data sample at the expense of selection efficiency. Less commonly, techniques which combine these physics variables in pre-determined ways are included. Genetic programming makes no presupposition on the final form of a solution. The charm photoproduction experiment FOCUS is an upgraded version of NAL-E687 which collected data using the Wideband photon beamline during the 1996–1997 Fermilab fixed-target run. The FOCUS experiment utilizes forward multiparticle spectrometer to study charmed particles produced by the interaction of high energy photons with a segmented BeO target. Charged particles are tracked within the spectrometer by two silicon microvertex detector systems. One system is interleaved with the target segments; the other is downstream of the target region. These detectors provide excellent separation of the production and decay vertices. Further downstream, charged particles are tracked and momentum analyzed by a system of five multiwire proportional chambers and two dipole magnets with opposite polarity. Three multicell threshold detectors are used to discriminate among electrons, pions, kaons, and protons. The experiment also contains a complement of hadronic and electromagnetic calorimeters and muon detectors.

To apply the genetic programming technique to FOCUS charm data, first step is performed by identifying a number of variables which may be useful for separating charm from non-charm. The variables which may separate the decays of interest from other charm decays are identified. Many of the variables selected are those that are used as cuts in traditional analyses, but since genetic programming combines variables not in *cuts*, but in an algorithmic fashion, a number of variables are also included, that may be weak indicators of charm or the specific decay in question.

Definition of Fitness

For the doubly Cabibbo suppressed decay $D^+ \to K^+\pi^+\pi^-$, very small signals are required. The fitness of a tree, as returned to the GPF, describes how small the error on a measurement of BR $D^+ \to K^+\pi^+\pi^-)$/BR $D^+ \to K^+\pi^+\pi^-)$ will be. Inorder to maximize the expected significance of the possible signal, S_{DCS}/sqrt($S_{DCS} + B_{DCS}$), where S_{DCS} and B_{DCS} are the doubly Cabibbo suppressed signal and background, respectively. However, with a small number of signal events and a very large search space, this method is almost guaranteed to optimize on fluctuations. Instead, one may choose to *predict* the number of signal events, S_{DCS}, from the behavior of the corresponding Cabibbo favored decay mode. In the test of this method on $D^+ \to K^+\pi^+\pi^-$, the PDG branching ratio was used to estimate the sensitivity. Assuming equal selection efficiencies, SCF (the signal yield in the Cabibbo favored mode) is proportional to the predicted number of doubly Cabibbo suppressed events. SCF is determined by a fit to the Cabibbo favored invariant mass distribution and B_{DCS} is determined by a linear fit over the background range (excluding the signal region) in the doubly Cabibbo suppressed invariant mass distribution. Since optimizing is based on B from the doubly Cabibbo suppressed mass plot, the user must be concerned about causing downward fluctuations in B_{DCS} which would *appear* to improve the significance. This is addressed in two ways. First, a penalty is applied to the fitness (or significance) based on the size of the tree; Second, the significance only on even-numbered events is optimized. Then the odd-numbered events can be considered to assess any biases. Because the genetic programming framework is set up to minimize, rather than maximize, the fitness and inorder to enhance differences between small changes in significance, the quantity is minimized as

$$\frac{S_{pred} + B_{DCS}}{S^2{}_{CF}} * 10000 * (1 + 0.005 * No\ of\ nodes)$$

where

$$S_{pred} = S_{CF} * \frac{BR(D^+ \to K^+\pi^+\pi^-)}{BR(D^+ \to K^+\pi^+\pi^+)}$$

The relative branching ratio is taken from the PDG. This penalty factor of 0.5% per node is arbitrary and is included to encourage the production of simpler trees. Thus it is found that at least 500 events are required, in Cabibbo favored and

doubly Cabibbo suppressed modes combined, are selected. It is also required that the Cabibbo favored signal be observed at the 6σ level. Both of these requirements ensure that a good Cabibbo favored signal exists. Trees which fail these requirements are assigned a fitness of 100, a very poor fitness.

Functions and Variables

The user can supply a wide variety of functions and variables to the GPF (Genetic Programming Functions) which can be used to construct trees. The constructed tree is evaluated for every combination. Events for which the tree returns a value greater than zero are kept. Fits are made to determine SCF and BDCS. Mathematical Functions and Operators: The first group of functions are standard mathematical (algebraic and trigonometric) functions and operators. Every function must be valid for all possible input values, so division by zero and square roots of negative numbers must be allowed. These mathematical functions and the protections used are shown in Table 8.2. If close agreement between simulation and data were deemed important, a term to ensure this could be added to the fitness definition.

Boolean Operators: The Boolean operators must take all floating point values as inputs. Thus the user may define any number greater than zero as "true" and any other number as "false." Boolean operators return 1 for "true" and 0 for "false." The **IF** operator is special; it returns the value of its second operand if the value of its first operand is true; otherwise it returns zero (or false). The comparison operator $<=>$ can also be used, as defined in the Perl programming language. This operator returns -1 if the first value is less than the second, $+1$ if the opposite is true, and 0 if the two values are equal. The Boolean operators are listed in Table 8.3. A large number of variables, or terminals, are also supplied. These can be classified into several

Table 8.2 Mathematical functions and operators. f(n) is the sigmoid function commonly used in neural networks

Operator	Description
$+$	
$-$	
$\times$	
/	Divide by $0 \to 1$
x^y	x is 1^{st} argument, y is second
log	$\log\lvert x\rvert,\ \log 0 = 0$
sin	
cos	
$\sqrt{}$	$\sqrt{\lvert x\rvert}$
neg	negative of x
sign	returns $(-1, 0, +1)$
$f(n)$	$1/(1+e^{-n})$
max	Maximum of two values
min	Minimum of two values

Table 8.3 Boolean operators and the comparison operator

Operator	Description
$>$	Greater than
$<$	Less than
AND	AND operator
OR	OR operator
NOT	Inversion operator
XOR	True if one and only one operand is true
IF	2^{nd} value if 1^{st} true, 0 if false
$<=>$	$>\rightarrow +1,\ <\rightarrow -1, =\rightarrow 0$

groups: 1) vertexing and tracking, 2) kinematic variables, 3) particle ID, 4) opposite side tagging, and 5) constants. All variables have been redefined as dimensionless quantities.

Vertexing Variables: The vertexing and tracking variables are mostly those used in traditional analyses, isolation cuts, vertex CLs and isolations, etc. The tracking variables which are calculated by the wire chamber tracking code are not generally used in analyses. The vertexing and tracking variables are shown in Table 8.4.

Kinematic Variables: Of the kinematic variables, most are familiar in HEP analyses. The most prominent exception is $\sum p_T^2$, which is the sum of the squares of the daughter momenta perpendicular to the *charmed parent* direction. When large, this

Table 8.4 Vertexing and Tracking variables

Variable	Units	Description
ℓ	cm	Distance between production and decay vertices
σ_ℓ	cm	Error on ℓ
ℓ/σ_ℓ	–	Significance of separation between vertices
ISO1, ISO2	–	Isolation of production and decay vertices
OoM	σ	Decay vertex out of target
POT	σ	Production vertex out of target
CLP, CLS	–	CL of production and decay vertices
σ_t	ps	Lifetime resolution

Table 8.5 Kinematic variables

Variable	Units	Description
$\#\tau$	–	Lifetime/mean Lifetime
P	GeV/c	Charm Momentum
p_T	GeV/c	p transverse to beam
$\sum p_T^2$	$\mathrm{GeV}^2/\mathrm{c}^2$	Sum of daughter p_T^2
m_{err}	$\mathrm{MeV}/\mathrm{c}^2$	Error of reconstructed mass
TS	0, 1	Early, late running
NoTS	0, 1	Opposite of TS

variable means the invariant mass of the parent particle is generated by large opening angles rather than highly asymmetric momenta of the daughter particles. In this category, binary variables NoTS and TS are also included, which represent running before and after the installation of a second silicon strip detector. The kinematic variables are shown in Table 8.5.

Particle Identification: For particle ID, the standard FOCUS variables are used for identifying protons, kaons, and pions. Boolean values are also included from the silicon strip tracking code for each of the tracks being consistent with an electron (zero-angle) and the maximum CL that one of the decay tracks is a muon. The particle ID input variables are shown in Table 8.6.

Opposite Side Tagging: Any cut on tagging methods is too inefficient to be of any use, but in combination with other variables, variables representing the probability of the presence of an opposite side decay may be useful. Three such variables were formed and investigated. The first attempts to construct charm vertices from tracks which are not part of the production *or* decay vertices. The best vertex is chosen based on its confidence level. The second general method of opposite side tagging is to look for kaons or muons from charm decays that are not part of the decay vertex. (Here, tracks in the production vertex are included since often the production and opposite side charm vertices merge.) The confidence level of the best muon of the correct charge and the kaonicity of the best kaon of the correct charge not in the decay vertex is determined. The variables for opposite side charm tagging are shown in Table 8.7.

Constants: In addition to these variables, a number of constants are also supplied. The user explicitly includes 0 (false) and 1 (true) and allow the GPF to pick real constants on the interval $(-2, +2)$ and integer constants on the interval $(-10, +10)$. The optimization in this example uses 21 are operators or functions and 34 terminals (variables or constants).

Table 8.6 Particle ID variables

Variable	Units	Description
$\Delta\pi K_1$	–	K not π
π_{con1}	–	π consistency, first pion
π_{con2}	–	π consistency, second pion
μ_{max}	–	Maximum μ CL of all tracks
Ke, π_{e1}, π_{e2}	0/1 (True/False)	Electron consistency from silicon tracker

Table 8.7 Opposite side tagging variables

Variable	Units	Description
CL_{opp}	–	CL of highest vertex opposite
$\Delta W(\pi K)_{opp}$	–	Highest kaonicity not in secondary
$CL\mu_{opp}$	–	Highest muon not in secondary

Anatomy of a Run

Once the fitness, functions and terminals are defined, the GPF is ready to start. To recap, the steps taken by the GPF are:

(1) Generate a population of individual programs to be tested.
(2) Loop over and calculate the fitness of each of these programs.
   (a) Loop over each physics event.
   (b) Keep events where the tree evaluates to > 0, discard other events.
   (c) For surviving events, fit the Cabibbo favored signal and doubly Cabibbo suppressed background.
   (d) Return the fitness calculated.
(3) When all programs of a generation have been tested, create another generation by selecting programs for breeding according to the selection method. Apply breeding operators such as cross-over, mutation, and reproduction.
(4) Every few generations, exchange the best individuals among "islands."
(5) Continue, repeating steps 2–4 until the desired number of generations is reached.

8.3.3 Selecting Genetic Programming Parameters

There are a large number of parameters that can be chosen within the GPF, such as numbers of individuals, selection probabilities, and breeding probabilities. Each of these can affect the evolution rate of the model and determine the probability that the genetic programming run doesn't have enough diversity to reliably find a good minimum. It should be emphasized, though, that the final result, given enough time, should not be affected by these choices (assuming sufficient diversity). The default parameters for the studies presented in this section are given in Table 8.8. In monitoring the test runs, the fitness and size of each individual is viewed. The user only

Table 8.8 Default genetic programming parameters for studies

Parameter	Value
Generations	6
Programs	1000
Sub-populations	20
Exchange Intreval	2 generations
Number Exchanged	5
Selection method	Fitness-over-select
Crossover Probability	0.85
Reproduction Probability	0.10
Mutation Probability	0.05
Generation method	Half grow, half full
Full depths	2–6
Maximum depth	17

considers individuals which have a fitness less than about 3 times worse than the fitness of a single-node tree which selects all events. This excludes individuals where no events were selected and others where much more signal than background was removed. Then the average and best fitness as a function of generation is observed while varying various parameters of the genetic programming run. Figure 8.5 shows the effects of various ways of doubling the total number of programs examined by the genetic programming run. The user begins with a run on 20 sub-populations with 1000 individuals per sub-population and lasting 6 generations. Then each of these quantities is doubled. One can see that either doubling the sub-population size or doubling the number of sub-populations gives about the same best and average fitness as the original case. However, doubling the number of generations has a significant effect on the average and best fitness of the population. After 12 generations (plus the initial generation "0"), evolution is still clearly occurring. In addition to changing the number of individuals evaluated during the genetic programming run as discussed above, there are a number of parameters in the genetic programming framework that can be adjusted. These may change the rate or ultimate end-point of the evolution.

Figure 8.6 shows the effect of changing some of the basic parameters of the genetic programming framework on the evolution of the population. These plots show the effect over 12 generations of changing the selection method, the number of programs exchanged during the migration phase, and the size of the initial programs generated. Exchanging 2 rather than 5 individuals from each process every 2

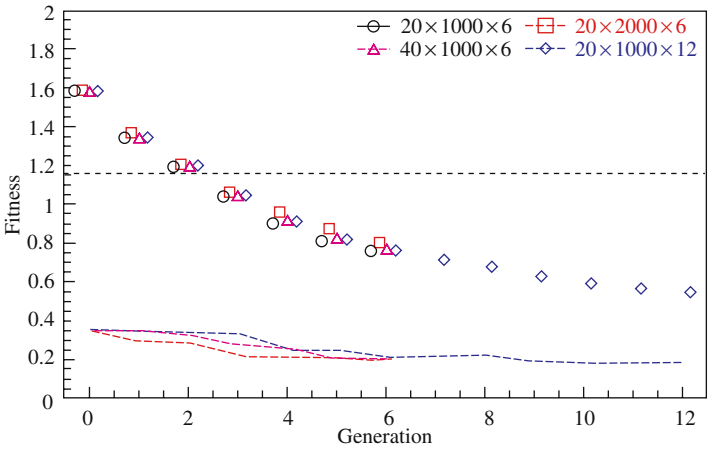


Fig. 8.5 Plots of fitness vs. generation for increased statistics. The points show the average fitness for a generation and the curves show the fitness of the best individual. The straight line shows the fitness of the initial sample. The circles and solid line show the initial conditions: 20 sub-populations, 1000 individuals per sub-population, and 6 generations. The squares and dashed line show the effect of doubling the individuals per sub-population to 2000. The triangles and dotted line show the effect of doubling the sub-populations to 40. The diamonds and dotted-dashed line show the effect of doubling the number of generations to 12

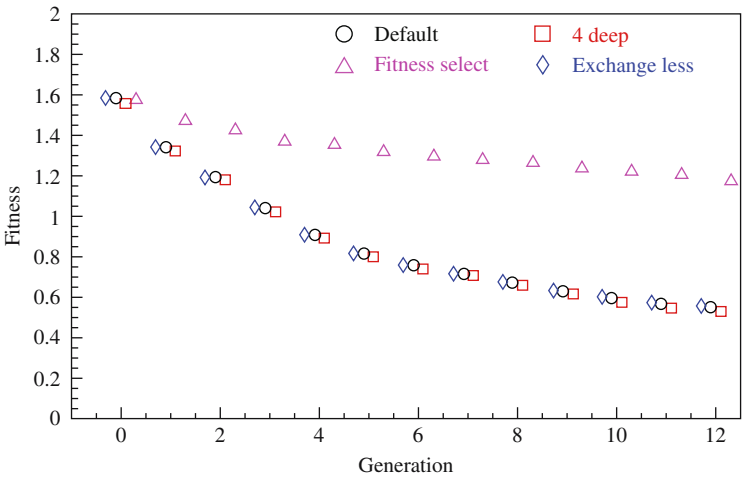


Fig. 8.6 Plots of fitness vs. generation for different GP parameters. The points show the average fitness for a generation. The circles show the default settings. The squares show the effect of changing the "full" part of the initial population generation to produce trees with depths of 2–4 (rather than the default 2–6). The diamonds show the effect of exchanging 2 rather than 5 individuals per process in the migration phase. The triangles show the effect of using "fitness" rather than "fitness over-select" selection. (The fitness over-select method is used in all other plots)

generations results in no change in the evolution rate. (For the standard case with 20 CPUs each with a population of 1000, exchanging 5 individuals from each of 20 CPUs means that 10% of the resulting sub-population is in common with each other sub-population.) Changing the selection method used has a dramatic effect on the evolution. Changing from the default "fitness-over-select" method to the more common "fitness" method results in a much more slowly evolving population. Recall that in the standard fitness selection, the probability of an individual being selected to breed is directly proportional to the fraction of the total fitness of the population accounted for by the individual.

In the fitness-over-select method, individuals in the population are segregated into two groups based on their fitness. Individuals which breed are selected more often from the more fit group (which also tends to be smaller). Selection within each group is done in the same way as for the standard fitness selection method. Conventional wisdom is that fitness-over-selection can be useful for large populations (larger than about 500 individuals). For smaller populations, there are risks associated with finding local minima. Half of the trees are generated by the "full" generation method. In the default evolution plot shown in Figure 8.6, the beginning depths of these trees are from 2 to 6. So, 10% of the initial trees are of depth 2, 10% are of depth 3, etc., up to 10% of depth 6. The remaining 50% are generated by the "grow" method. As can be seen, changing the depth of the full trees so that trees of depth 5 and 6 are not allowed has little effect on the evolution rate. Other, earlier

studies indicate that this change may positively affect the fitness in early generations and negatively affect the fitness of the later generations.

Figure 8.7 shows the effect of changing the mutation probabilities. Increasing the mutation probability from 5% to 10% (at the expense of the crossover probability) does very little to change the evolution rate. Reducing the mutation probability to 2.5% or even 0% has a similarly small effect. However, at higher mutation rates, the user worries about the effects of too high a mutation rate on later generations where stability should be increasing. Figure 8.8 shows the effect of reducing the number of functions used in the search. In the first case, a number of the algebraic, trigonometric, logical, and arithmetic operators were removed. Operators such as log, x^y, sin, cos, XOR, and others were not allowed to be used. This reduces the diversity of the program set, but may allow a minimum to be found faster if those operators are not necessary. In another trial, the variables not normally used in FOCUS analyses, such as opposite side tagging, track CLs, muon and electron mis-ID for hadrons, etc were removed. In a final trial both non-standard variables and questionable functions were removed. The assumption in all three cases is the same: if the functions or variables added to the minimal case do not positively contribute to the overall fitness, slower evolution and possibly a worse final solution should be observed since the useful parameter space is less well covered. That this is not observed in the average case (which is less susceptible to fluctuations) suggests that the "extra" functions and variables are useful and ultimately better solutions may be found by including them.

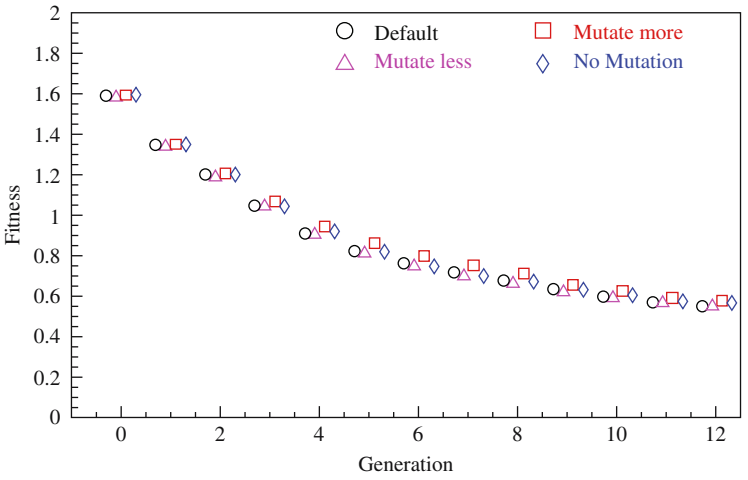


Fig. 8.7 Plots of fitness vs. generation for different mutation rates. The points show the average fitness for a generation. The circles and show the default settings. The squares show the effect of doubling the mutation probability from 5% to 10%, the triangles show the effect of halving the mutation probability from 5% to 2.5%, and the diamonds show the effect of eliminating mutation

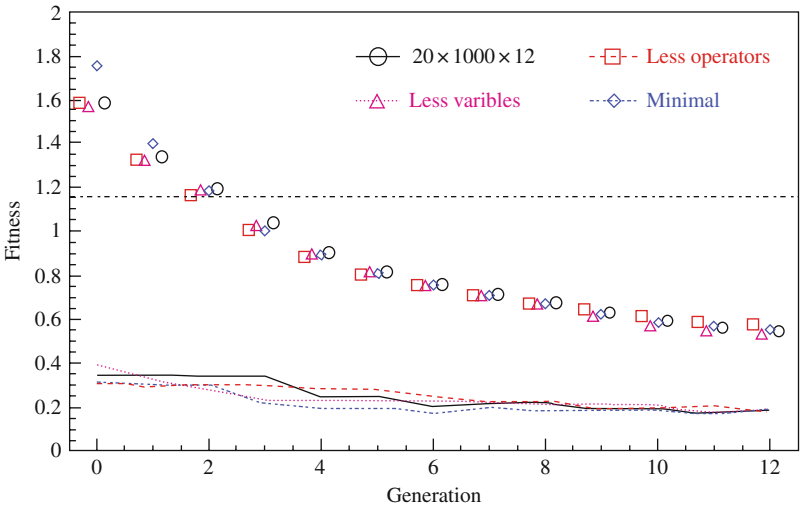


Fig. 8.8 Plots of average and best fitness vs. generation for sets of reduced functions. For "Less operators," (squares and dashed line) the set of arithmetic, algebraic, and logical operators was reduced to a minimal set. For "Less Variables" (triangles and dotted line) only "standard" analysis variables were included. For "Minimal" (diamonds and dashed-dotted line) both reductions were employed

8.3.4 Testing Genetic Programming on $D^+ \to K^+\pi^+\pi^-$

Having explored various settings in the GPF to determine the way to obtain the best evolution of programs, the ultimate performance and accuracy is investigated. The GPF attempting to minimize the quantity is

$$\frac{B_{DCS} + S_{pred}}{S^2_{CF}} * 10000^*(1 + 0.005^* No\ of\ nodes)$$

where B_{DCS} is a fit to the background excluding the $\pm 2\sigma$ signal window and Spred is the expected doubly Cabibbo suppressed yield determined from the PDG value for the $D^+ \to K^+\pi^+\pi^-$ relative branching ratio and the Cabibbo favored signal (S_{CF}).

- $\pi_{con} > -8.0$ for both pions
- $\Delta W(\pi K) > 1.0$ for kaon
- $1.75\ GeV/c^2 < Mass < 1.95\ GeV/c^2$

The initial data sample is shown in Figure 8.9. The Cabibbo favored signal dominates, and the level of doubly Cabibbo suppressed candidates (the linear histogram) is higher than the Cabibbo favored background. The Cabibbo favored fit finds $253\,180 \pm 660$ events. The $D^+ \to K^+\pi^+\pi^-$ and $D^+ \to K^+\pi^+\pi^+$ events are selected using the genetic programming technique with the parameters listed in Table 8.9. These parameters are similar to those used in the earlier studies, but the programs per generation and the number of generations are increased. Figure 8.10 shows the data

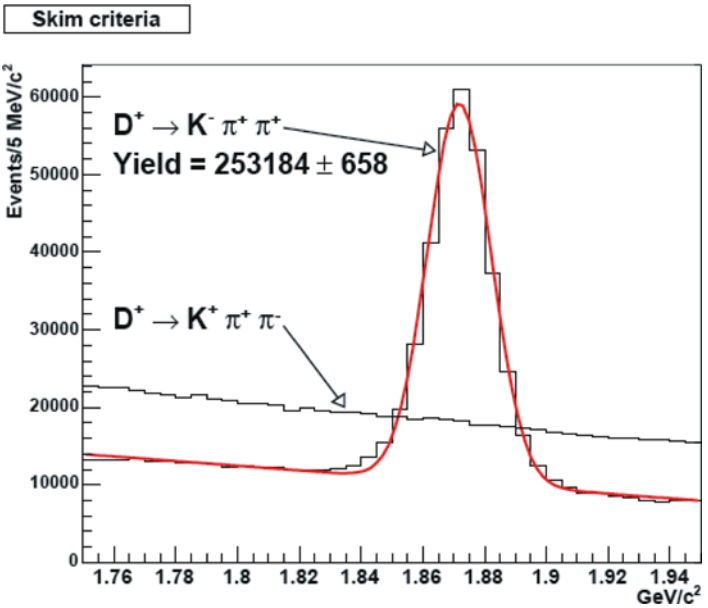


Fig. 8.9 The initial $D^+ \rightarrow K^+\pi^+\pi+$ and $D^+ \rightarrow K^+\pi^+\pi^-$ candidate distributions (the cuts are described in the text). The linear doubly Cabibbo suppressed invariant mass distribution is superimposed on the Cabibbo favored distribution

after selection by the GPF for the Cabibbo favored and doubly Cabibbo suppressed decay modes. The GPF ran for 40 generations. A doubly Cabibbo suppressed signal is now clearly visible. $62\,440 \pm 260$ (or about 25%) of the original Cabibbo favored events remain and 466 ± 36 doubly.

Table 8.9 Genetic Programming parameters for $D^+ \rightarrow K^+\pi^+\pi^-$ Optimization

Parameter	Value
Generations	40
Programs/sub-population	1500
Sub-populations	20
Exchange interval	2 generations
Number exchanged	5
Selection method	Fitness-over-select
Cross-over probability	0.85
Reproduction probability	0.10
Mutation probability	0.05
Generation method	Half grow, half full
Full depths	2–6
Maximum depth	17

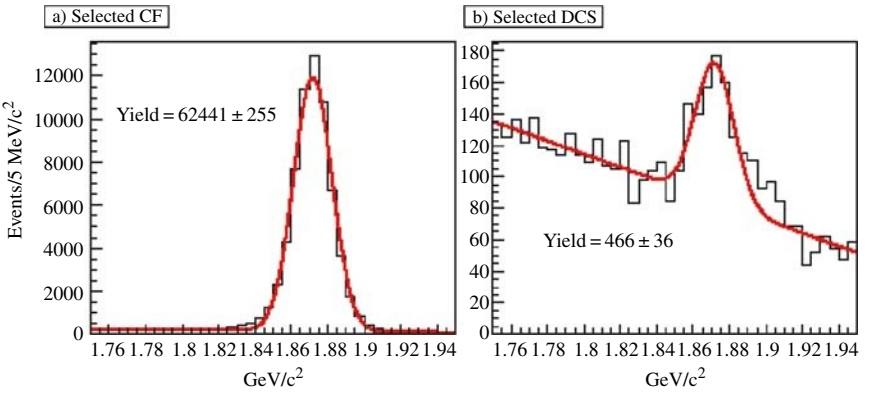


Fig. 8.10 $D^+ \to K^+\pi^+\pi^+$ (a) and $D^+ \to K^+\pi^+\pi^-$ (b) signals selected by genetic programming

The free parameters are the yield and the background shape and level. The evolution of the individuals in the genetic programming is shown in Figure 8.11. In addition to the variables plotted before, the average size for each generation is also plotted. It is found that the average size reaches a minimum at the 4th generation, nearly plateaus between the 20th and 30th generations, and then begins increasing again and is still increasing at the 40th generation. In Figure 8.11, the average and best fitnesses seems to stabilize, but in Figure 8.12 an enlargement of the best fitness for later generations is seen. From this plot, it is apparent that evolution is still occurring at the 40th generation and new, better trees are still being found.

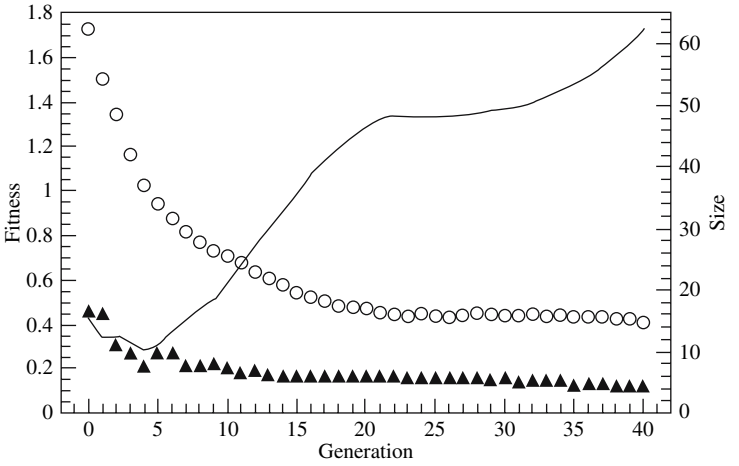


Fig. 8.11 Evolution of $D^+ \to K^+\pi^+\pi^+$ programs. The open circles show the average fitness of the population as a function of generation. The triangles show the fitness of the best individual. The solid line shows the average size of the individuals as a function of the generation

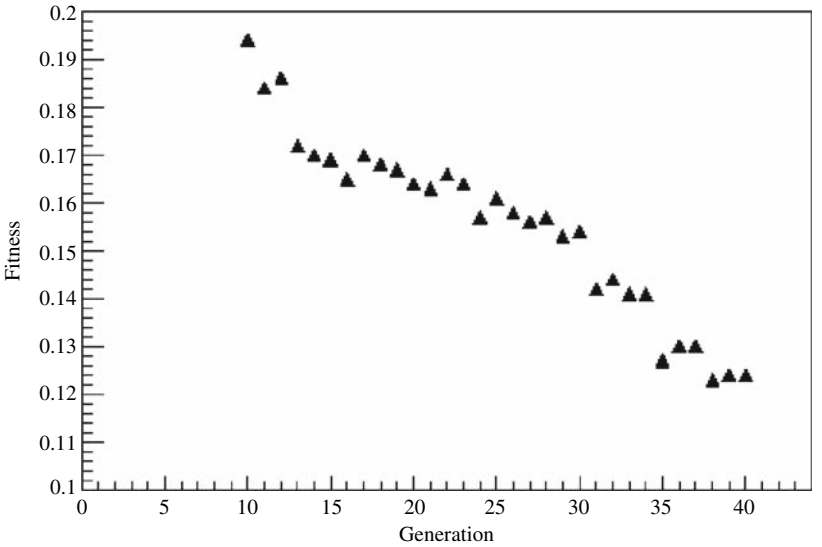
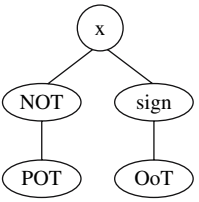


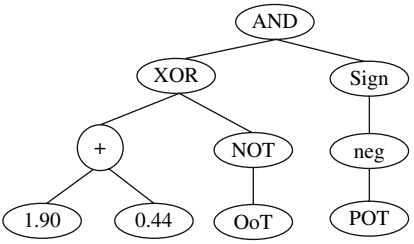
Fig. 8.12 Expanded view of evolution of $D^+ \rightarrow K^+\pi^+\pi^+$ programs. The triangles show the fitness of the best individuals for later generations, the same data as in Figure 8.11, but on an expanded scale

Example trees from various generations Figure 8.13 shows four of the most fit trees from the initial generation (numbered 0); no evolution has taken place at this point. It is interesting to note several things. First, the best trees are all rather small, although the average tree in this generation is over 15 nodes in size. Second, the most fit tree in this generation (a) is recognizable to any FOCUS collaborator: it requires that the interaction vertex point be in the target and that the decay vertex point be located outside of target material. The second most fit tree (b) is algorithmically identical to the first, but has a slightly worse fitness because it is considerably larger than the first tree. Tree (c) is nearly identical in functionality to (a) and (b) but actually does a slightly better job of separating events than (a) or

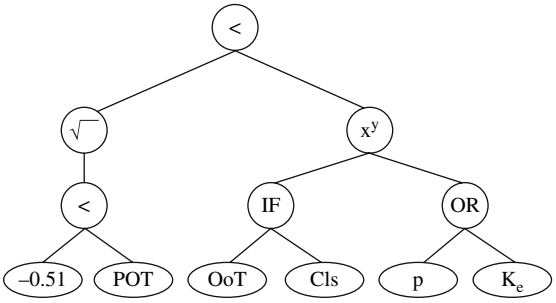
Its fitness is slightly worse because of its larger size (12 nodes vs. 5 and 10 nodes respectively). The four best individuals from generation 2 are shown in Figure 8.14. A few observations are in order. First, the most fit tree (a) doesn't include either of the elements found in the best trees in generation 0 (OoT and POT), but this tree is significantly more fit than any tree in generation 0. The second best tree (b) does include these elements, mixed in with others. Also, one can see that the fourth most fit tree (d) is quite large (44 nodes) compared to the other best trees in the generation. As shown in Figure 8.11, the average size tree of the population reaches a minimum in generation 4. One possible interpretation of this behavior is that at this point, the GPF has determined the parts of successful solutions. Going forward, it is assembling these parts into more complex solutions. The three best solutions from generation four are shown in Figure 8.15. The best solution from generation four (a) is very similar to the best solution from generation two, but with the two end



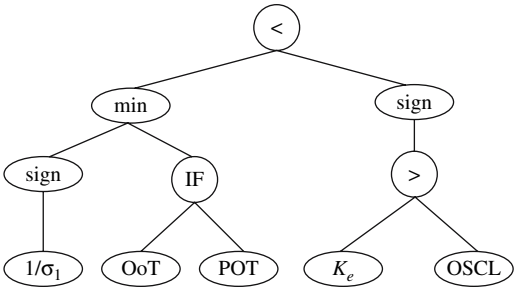
(a) Fitness 0.458



(b) Fitness 0.469



(c) Fitness 0.471

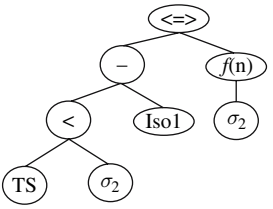


(d) Fitness 0.472

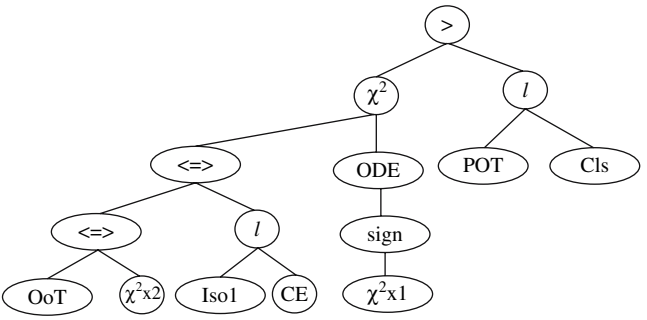
Fig. 8.13 The four most fit trees from generation 0 of a $D^+ \to K^+\pi^+\pi^-$ run

nodes replaced with new end nodes. The second and third best solutions are clearly related, with just one small difference: the second best tree (b), with the sub-branch: OS_μ <=> OoT only lets about 250 events with OoT through the filter while the third best tree (c) allows about 9300 such events through.

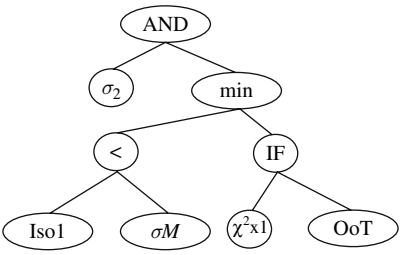
In Figure 8.16, it is found that tree (a) and tree (b) have some pieces in common, but are quite different. They are also about equally good at separating background from signal, but tree (b) has the larger fitness due to its larger size.



(a) Fitness 0.305



(b) Fitness 0.309

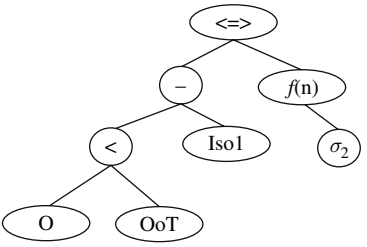


(c) Fitness 0.383

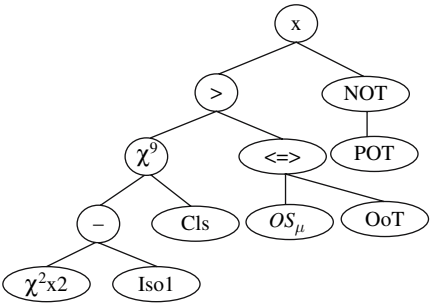
Fig. 8.14 The four most fit trees from generation 2 of a $D^+ \to K^+\pi^+\pi^-$ run

8.3.5 Concluding Remarks

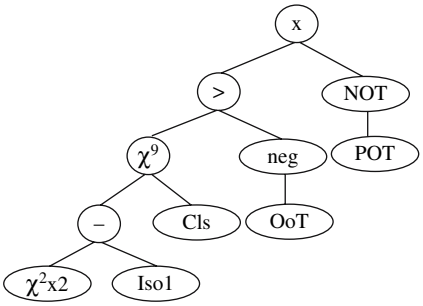
Thus an appropriate introduction to genetic programming and its application in high energy physics analyses was discussed. The use of the technique in separating the doubly Cabibbo suppressed decay $D^+ \to K^+\pi^+\pi^-$ from the copious background was demonstrated and also proved that this technique can improve upon more traditional analysis techniques. As with any analysis technique, care must be taken to understand the possible systematic errors introduced by the technique.



(a) Fitness 0.211



(b) Fitness 0.272

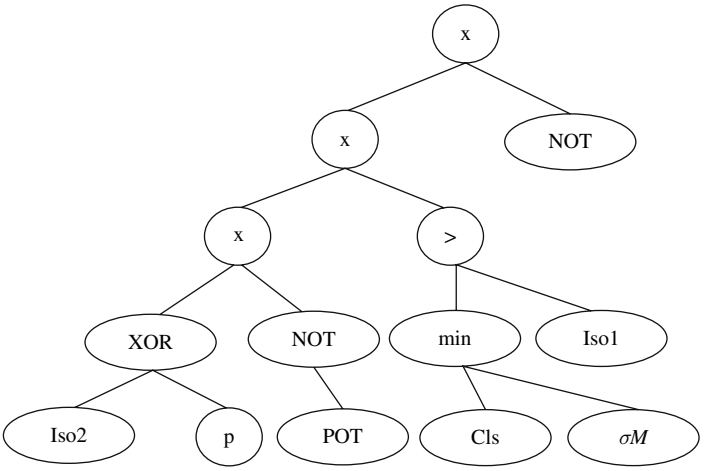


(c) Fitness 0.281

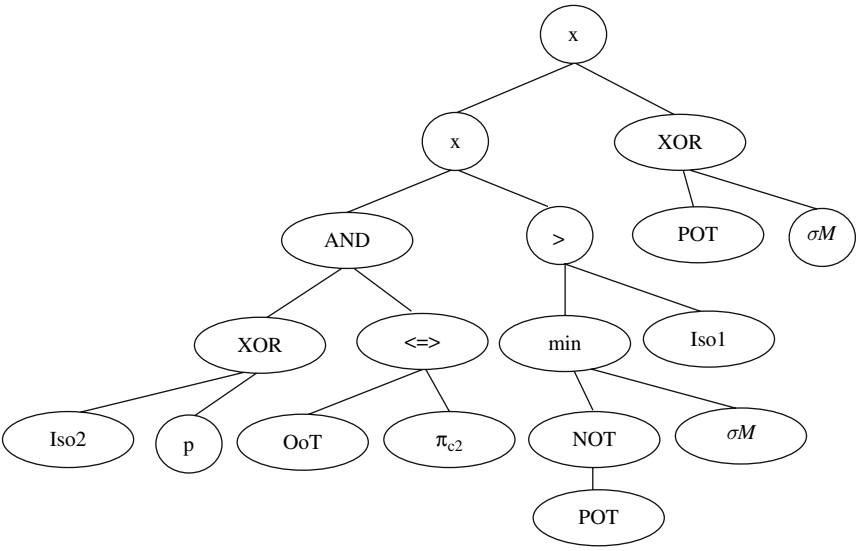
Fig. 8.15 The four most fit trees from generation 4 of a $D^+ \rightarrow K^+\pi^+\pi^-$ run

8.4 Using Genetic Programming to Generate Protocol Adaptors for Interprocess Communication

In the field of evolvable computing, software (and hardware) is developed that adapts itself to new runtime environments as necessary. The runtime environments targeted in this application are open distributed systems in which interprocess communication forms an essential problem. In these environments an application consists of processes that communicate with other processes to each specific goal. With the advent of mobile devices these processes do not necessarily know in which kind of runtime environments they will execute. Therefore they rely on standardized solutions, such as JINI, to find other processes offering certain behavior. Once the other process is known, the real problems start. How can the requesting process communicate with the unknown offered process? Given the fact that those processes are developed by different organizations, the protocols provided and required can vary greatly. As a result protocol conflicts arise. On first sight, a solution to this problem would be to offer protocol adaptors between every possible pair of processes. The problem with this approach is that the number of adaptors grows quadratic to the number of process protocols and as such it simply doesn't scale. The solution is



(a) Fitness 0.194



(b) Fitness 0.198

Fig. 8.16 The four most fit trees from generation 10 of a $D^+ \to K^+\pi^+\pi^-$ run

to automate the generation of protocol adaptors between communicating processes. As a potentially useful technique for this adaptor generation, the research domain of adaptive systems is explored. It was found that the combination of genetic programming, classifier systems, and a formal specification in terms of Petrinets allowed automating the detection of protocol conflicts, as well as the creation of program code or adaptors that solve these conflicts. This application reports on an experiment that was performed to validate this claim.

8.4.1 Prerequisites of Interprocess Communication

Processes communicate with each other only by sending messages over a communication channel. Communication channels are accessed by the process' ports. Processes communicate asynchronously and always copy their messages completely upon sending. The connections between processes are full duplex: every process can send and receive messages over a port. This brings us in a situation where a process provides a certain protocol and requires a protocol from another process. A process can have multiple communication channels: for every communication partner and for every provided/required protocol. Other requirements were imposed on the interprocess communication to generate adaptors:

1. Implicit addressing. No process can use an explicit address of another process. Processes work in a connection-oriented way. The connections are set up solely by one process: the connection broker. This connection broker will also evolve adaptors and place them on the connections when necessary.
2. Disciplined communication. No process can communicate with other processes by other means than its ports. Otherwise, 'hidden' communication (e.g., over a shared memory) cannot be modified by the adaptor. This also means that all messages passed over a connection should be copied. Messages cannot be shared by processes (even if they are on the same host), because this would result in a massive amount of concurrency problems.
3. Explicit protocol descriptions. While humans prefer a protocol description written in natural language, computers need an explicit formal description of the protocol semantics. A simple syntactic description is no longer suitable.

8.4.2 Specifying Protocols

As a running example, a typical problem of communicating processes is chosen. Typically, a server provides a con currency protocol that can be used by clients. The clients have to adhere to this specification or they will not function. Since the client also expects certain concurrency behaviour from the server, it is possible that the required interface and provided interface differ. For example, a client/server can require/provide a full-fledged optimal transaction protocol or it can require/provide

a simple locking protocol. When two such protocols of a different kind interact, an incompatibility problem arises. In this example, a simple locking protocol of the server with which a client can typically lock a resource and then use it is considered. The API (Application Protocol Interface) for the server is described as follows. (A similar protocol description can be given for the clients.)

```
incoming lock(resource)
outgoing lock_true(resource)
outgoing lock_false(resource)
// lock_true or lock_false are sent back whenever
   a lock
// request comes in: lock_true when the resource
   is locked,
// lock_false when the resource couldn't be locked.
incoming unlock(resource)
outgoing unlock_done(resource)
// will unlock the resource. Send unlock_done back when
   done.
incoming act(resource)
outgoing act_done(resource)
// will do some action on the process.
```

The semantics of this protocol can be implemented in different ways. Two kinds of locking semantics: Counting semaphores allow a client to lock a resource multiple times. Every time the resource is locked the lock counter is increased. If the resource is unlocked the lock counter is decreased. The resource is finally unlocked when the counter reaches zero. Binary semaphores provide a locking semantics that doesn't offer a counter. It simply remembers who has locked a resource and doesn't allow a second lock. When unlocked, the resource becomes available again. Differences in how the API considers lock and unlock can give rise to protocol conflicts. In Figure 8.17 the client process expects a counting semaphore from the server process, but the server process offers a binary semaphore. The client can lock a resource twice and expects that the resource can be unlocked twice. In practice the server just marked the resource is locked. If the client unlocks the resource, the resource will be unlocked. This protocol conflict arises because the API does not specify enough semantic information. Hence, a more detailed and generally applicable formalism, namely Petrinets is proposed, to offer an explicit description of the protocol semantics. Petrinets offer a model in which multiple processes traverse states by means of state transitions. In the context of this locking example, this allows the user to write a suitable adaptor by relying on: (1) which state the client process expects the server to be in; (2) in which state the server process is. Both kinds of information are essential:

As an example Petrinet that offers the needed semantics, the left part of Figure 8.18 specifies a binary semaphore locking strategy. The current state is un-locked. From this state the process requiring this protocol, can choose only one

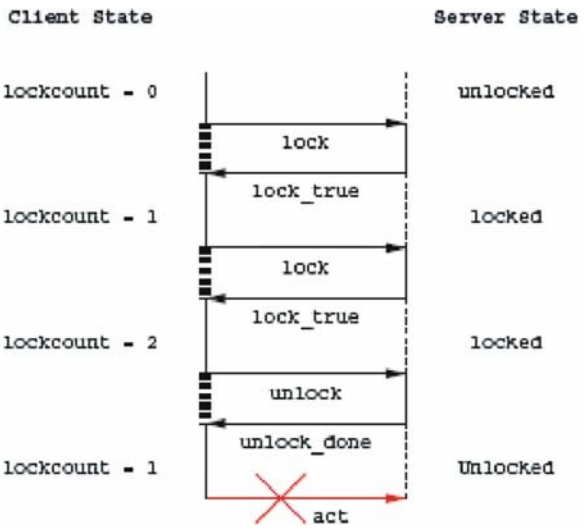


Fig. 8.17 Conflict between a counting semaphore protocol and a binary semaphore protocol

action: lock. It then goes to the locking state until lock true or lock false comes back. This Petrinet can also be used to model the behaviour of a process that provides this protocol. It is perfectly possible to offer an protocol that adheres to this specification, in which case, the incoming lock is initiated from the client, and lock true or lock false is sent back to the client hence making the transition.

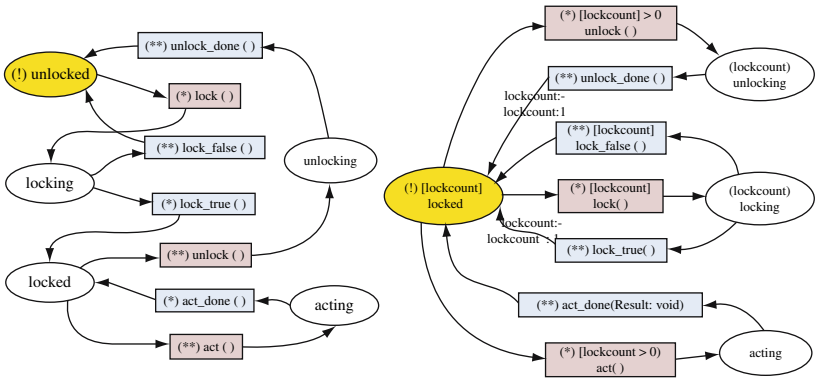


Fig. 8.18 Two Petri-net descriptions of process protocols. Ellipses correspond to states. Rectangles correspond to transitions. The current state (marked with '!') is coloured yellow. The red transitions (marked with '*') represent incoming messages. The blue ones (marked with '**') represent outgoing messages

8.4.3 Evolving Protocols

Protocol adaptors overcome the semantic differences between two processes. Classifier systems are known to work very well in cooperation with genetic algorithms, they are Turing complete and they are symbolic. This is important because the Petri net description is in essence a symbolic representation of the different states in which a process can reside. If this representation would be numerical, techniques such as neural networks, reinforcement learning and Q-learning would probably be used. The standard questions before implementing any genetic programming technique are: What are the individuals and their genes? How are the individuals represented? How the fitness of an individual is defined or measured? How to create individuals initially? How is mutation and cross-over of two individuals performed? How to compute a new generation from an existing one? In this implementation, the individuals will be protocol adaptors between communicating processes. The question of how to represent these individuals is more difficult. Well-known programming languages can be used to represent the behaviour of the adaptor. Unfortunately, the inevitable syntactic structure imposed by these languages complicates the random generation of programs. Moreover, these programming languages do not offer a uniform way to access memory.

An alternative that is more suitable for this purpose is the Turing complete formalism of classifier systems. A classifier system is a kind of control system that has an input interface, a finite message list, a classifier list and an output interface. The input and output interface put and get messages to and from the classifier list. The classifier list takes a message list as input and produces a new message list as output. Every message in the message list is a fixed-length binary string that is matched against a set of classifier rules. A classifier rule contains a number of possibly negated) conditions and an action. These conditions and actions form the genes of each individual in the genetic algorithm. Conditions and actions are both ternary strings (of 0, 1 and #). #' is a pass-through character that, in a condition, means 'either 0 or 1 match's. If found in action, it is simply replaced with the character from the original message. Table 8.10 shows a very simple example. When evaluating a

Table 8.10 Illustration of how actions produce a result when the conditions match all messages in the input message list. ~ is negation of the next condition. A disjunction of two conditions is used for each classifier rule. The second rule does not match for input message 001. The third rule does not match because the negated condition is not satisfied for input message 001

Input message list = {001, 101, 110, 100}

Condition		Action	Matches	Result
00#	101	111	yes	111
01#	1##	000	no	/
1##	~00#	###	no	/
1##	###	1#0	yes	100, 110

Output message list = { 111, 100, 110 }

classifier system, all rules are checked (possibly in parallel) with all available input messages. The result of every classifier evaluation is added to the end result. This result is the output message list.

A classifier system needs to reason about the actions to be performed based on the available input. In this implementation, the rules of a classifier system consist of a ternary string representation of the client state and server state (as specified by the Petri net), as well as a ternary string representing the requested Petri net transition from either the client process or the server process. With these semantics for the classifier rules, translating a request from the client to the server requires only one rule. Another rule is needed to translate requests from the server to the client (Table 8.11). The number of bits needed to represent the transitions depends on the number of possible state transitions in the two Petri nets. The number of bits needed to represent the states of each Petri net depends on the number of states in the Petri net as well as the variables that are stored in the Petri net. Although this is a simple example, more difficult actions can be represented. Consider the situation where the client uses a counting-semaphores locking strategy and the server uses a binary-semaphores locking strategy. In such a situation it is not required to send out the lock-request to the server if the lock count is larger than zero. Table 8.12 shows the representation of such a behaviour. The genetic programming implemented uses a full classifier list with variable length. The classifier list is an encoding of the Petrinets, as representation for the individuals. Every individual is initially empty. Every time an individual encounters a situation where there is no matching gene a new gene (i.e., a new classifier rule) will be added with a condition that covers this situation and a random action that is performed on the server and/or the client.

This way of working, together with the use of Petri nets guarantees that the genetic algorithm will only search within the subspace of possible solutions. Without these boundaries the genetic algorithm would take much longer to find solution. Fitness of an individual is measured by means of a number of test scenarios. Every test scenario illustrates a typical behaviour the client requests from the server. The fitness of an individual is determined by how many actions the scenario can

Table 8.11 Blind translation between client and server processes. The last 5 characters in column 1 represent the corresponding transition in the Petri net. The characters in the second and third column represent the states of the client and server Petri net, respectively. The fourth column specifies the action to be performed based on the information in the first four columns

Classifier condition			Action	Rule description
Requested transition	Client state	Server state	Performed action	
00####	####	###	11#...#	Every incoming action from the client (00) is translated into an outgoing action on the server (11)
01####	####	###	10#...#	Every incoming action from the server (01) is translated into an outgoing action to the client (10)

Table 8.12 Translating a client process lock request to a server process lock action when necessary

Classifier condition			Action	Rule description
Requested transition	Client state	Server state	Performed action	
00 001	##00	###	10 010…	If the client wants to lock (001) and already has a lock (~##00) we send back a lock_true (010)
00 001	##00	###	11 001…	If the client wants to lock (001) and has no lock (##00) we immediately send the message through (001).

execute without yielding unexpected behaviour. To avoid this kind of behaviour the proposed algorithm provides a covert channel that is used by the test scenario to contact the server to verify its actions. The genetic programming uses a steady-state GA, with a ranking selection criterion: compute a new generation of individuals, 10% of the individuals with the best fitness are reproduced. 10% of the worst individuals (not fit enough) are thrown away and cross-overs from the 10% best group are added. To create a crossover of individuals iteratation over both classifier lists is performed and each time a rule is randomly selected that will be stored in the resulting individual. It should be noted that the individuals that take part in cross-over are never mutated. The remaining 80% of individuals are mutated, which means that the genes of each individual are changed at random: for every rule, a new arbitrary action to be performed on server or client is chosen. On top of this, in 50% of the classifier rules, one bit of the client and server state representations is generalized by replacing it with a #. This allows the genetic program to find solutions for problems that are not presented yet. The parameters and their values used in this application is shown in Table 8.13.

Table 8.13 Parameters and their values

parameter	value
individuals (genotype)	variable-length classifier system represented as bitstring
population size	100
maximum generations (100 runs)	11
parent selection	ranking selection (10% best)
mutation	bitflip on non ranked individuals
mutation rate	0.8
crossover	uniform
crossover rate	0.1
input/output interfacing	Petri net state/transition representation
actions	message sending
fitness	number of successfully executed actions

8.4.4 The Experiment

The experiment is presented that shows the feasibility of the above techniques to automatically learn an adaptor between incompatible locking strategies. The experiment set up is a connection broker between two processes. The first process contacts the second by means of the broker. Before the broker sets up the connection it will generate an adaptor between the two parties to mediate semantic differences. It does so by requesting a running test process from both arties. The client will produce a test client and test scenarios. The server will produce a test server. In comparison with the original process, these testing processes have an extra testing port. Furthermore, this testing port is also used as the covert channel for validating actions at the server. The genetic program, using a set of 100 individuals (i.e., adaptor processes), will deploy the test processes to measure the fitness of a particular classifier system. Only when a perfect solution is reached, i.e., a correct adaptor has been found, the connection is set up. For reliability reasons the experiment is repeated (i.e., the execution of the genetic program) 100 times. The scenarios offered by the client are the ones that determine that kind of classifier system is generated. This process has been tried with three scenarios, as illustrated on the left of Figure 8.19. Scenario 1 is a sequence: [lock(), act(), unlock()]. Scenario 2 is the case explained in Figure 8.17. Scenario 3 is similar to scenario 1: [lock(), act(), act(), unlock()]. In all three scenarios, the same list of messages is issued three times to ensure that the resource is unlocked after the last unlock operation.

Observations

An examination of the results of several runs of genetic programming algorithm lead to the following observations: When the covert channel was used to measure

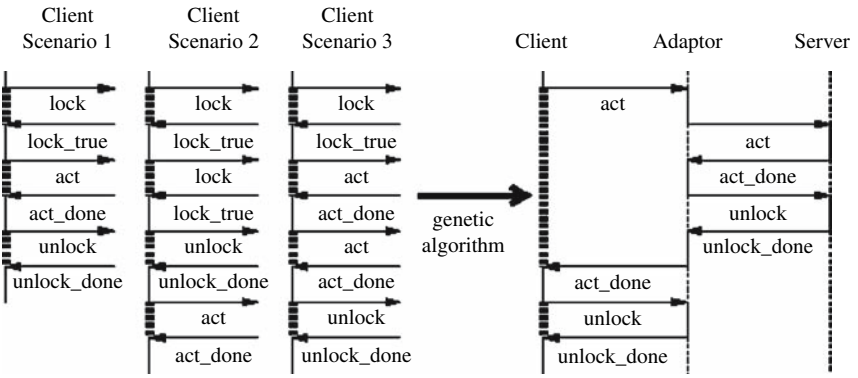


Fig. 8.19 The three test scenarios as initial input to the genetic programming. The dashed vertical lines are waits for a specific message (e.g., lock true). When using only scenarios 1 and 2 as input, one of the generated adaptors behaved as specified on the right hand side

the fitness, it was found that for all 100 runs of the GA a perfect solution was found within at most 11 generations. When the covert channel was not used to check the actions at the server side, the genetic algorithm often 30% created a classifier that doesn't even communicate with the server. In such a situation the classifier immediately responds lock true whenever the client requests a lock. Occasionally a classifier system was generated with a strange behaviour. Its fitness was 100%, but it worked asynchronously. In other words, the adaptor would contact the server process before the client process even requested an action from the server process. It could do so because the adaptor knew that the client would request that particular action in the given context. It implies that a learned algorithm can anticipate certain kinds of future behaviour. Initially, scenario 1 and 2 were only used to measure the fitness of each individual. The problem encountered was that sometimes, the adaptor anticipates too much and after the first act, keeps on acting. This problem was solved by assigning zero fitness to such solutions. As a last experiment the fitness was measured by combining information from all three scenarios. This allowed the genetic algorithm to find a perfect solution with less generations because separate individuals that developed behaviour for a specific test scenario were combined n a later generation using cross-over. This illustrates the necessity for a cross-over operator. A random search would take considerably more time to find a combination of both behaviours. One of the classifier systems that is generated by the genetic algorithm, when providing as input all three test scenarios, is given in Table 8.14. The produced classifier system simply translates calls from client to server and vice versa, unless it is about a lock call that should not be made since the server is already locked. The bit patterns in the example differ slightly from the bit patterns explained earlier. This is done to make a distinction between a 'transition- message' and a 'state-message'. All transition messages start with 00 and all state-messages start with 10 for client-states and 11 for server-states.

Discussion

In this approach, the problem of 'writing correct adaptors' is shifted to the problem of 'specifying correct test sets': whenever the developer of a process encounters an incompatibility, he needs to specify a new test scenario that avoids this behaviour. This test scenario is given as additional input to the genetic algorithm, so that the algorithm can find a solution that avoids this incompatibility. The result of this approach is that the programmer does not have to implement the adaptors directly, but instead has he responsibility of writing good test sets (i.e., a consistent set of scenarios that is as complete s possible). This is a non- trivial problem, since the test set needs to cover all explicit as well as implicit requirements. The main advantage of test sets over explicit adaptors is that a new adaptor may be needed for every pair of communicating processes, while only one test set is needed for each individual process. As such, test sets are more robust to changes in the environment: then a process needs to communicate with a new process, there is a good chance that the test set will already take into account potential protocol conflicts. Another

Table 8.14 The generated classifier system for a single run

Requested transition	Client state	Server state	Performed action	Description
00 00 01	10 00 #1	11 ####	00 10 01	client?lock() & client = locked → client.lock_true()
00 00 01	10 00 1#	11 ####	00 10 01	client?lock() & client = locked → client.lock_true()
00 00 01	10 00 00	11 010	00 11 01	client?lock() & client ≠ locked → server.lock()
00 00 10	10 00 1#	11 ####	00 10 11	client?unlock() & clientlock > 2 → client.unlock_done()
00 00 10	10 00 01	11 ####	00 11 10	client?unlock() & clientlock = 1 → server.unlock()
00 00 11	10 #####	11 ####	00 11 11	client?act() → server.act()
00 01 10	10 #####	11 ####	00 10 10	server?act_done() → client.act_done()
00 01 00	10 #####	11 ####	00 10 00	server?lock_false() → client.lock_false()
00 01 01	10 #####	11 ####	00 10 01	server?lock_true() → client.lock_true()
00 01 11	10 #####	11 ####	00 10 11	server?unlock_done() → client.unlock_done()
00 00 01	10 00 00	11 010	00 10 00	client?lock() & server = locked & client ≠ locked → client.lock_false()

important advantage of test sets is that they can help in automatic program verification. Bugs in the formal specification can be detected and verified at runtime. As such, this approach helps the developer to stay conform to the program specification. This clearly helps him in his goal to write better software.

8.4.5 Concluding Remarks

An automated approach to create intelligent protocol adaptors to resolve incompatibilities between communicating processes was proposed. Such an approach is indispensable to cope with the combinatorial explosion of protocol adaptors that are needed in an open distributed setting here processes interact with other processes in unpredictable ways. This approach uses a genetic programming technique that evolves classifier systems. These classifier systems contain classifiers that react upon the context they receive from both client process and server process. The context is defined as a combination of the possible client-side and server-side states as given by a user-specified Petri net. To measure the fitness of an adaptor, the user

needs to provide test scenarios as input. This enables the user to avoid undesired behavior in interprocess communication.

8.5 Improving Technical Analysis Predictions: An Application of Genetic Programming

As an approach to financial forecasting, technical analysis is based on the belief that historical price series, trading volume, and other market statistics exhibit regularities. There are two general approaches in technical analysis: one involves qualitative techniques and the other quantitative techniques. The qualitative techniques rely on the interpretation of the form of geometric patterns such as *double bottoms, head-and-shoulders*, and *support and resistance levels*; whilst the quantitative techniques try to create indicators such is *moving average* (MV*), relative strength indicators* (RSI), etc. Nonetheless, both techniques can be characterized by appropriate sequences of local minima and/or maxima. According to the peak form of *efficient market hypothesis* (EMH), since historical price information is already reflected in the present price, technical analysis is totally useless for predicting future rice movements. In recent years, however, this hypothesis has been directly challenged by a fair amount of studies, which supply evidence of predictability of security return from historical price patterns. Evidence in support of technical analysis can be classified into two categories: (1) evidence on systematic dependencies in security return. Researchers have reported positive or negative series correlation in returns of individual stocks or indices on various time period basis (e.g. daily, weekly, several monthly or yearly); (2) evidence on the returns earned by technical rules, including momentum and contrarian strategies, moving average rules and trade range breakout rules. It is not the purpose here to provide theoretical or empirical justification for the technical analysis. The aim of this study is to show how genetic programming (GP), a class of algorithms in evolutionary computation, can be employed to improve technical rules. The approach is demonstrated in a particular forecasting task based on the Dow Jones Industrial Average (DJIA). Quantitative technical rules are often used to generate buy or sell signals based on each rule interpretation. One may want to use technical rules to answer questions such as "*is today is a good time to buy if I want to achieve a return of 4% or more within the next 63 trading days?*" and "*is today the right time to sell if I want to avoid a lost of 5% or more within the next 10 days?*". However, the way Improving Technical Analysis Predictions technical rules are commonly used may not be adequate to answer these questions. How to efficiently apply them and adapt them to these specific prediction problems is a non-trivial task. Thus a GP approach is proposed that is capable of combining individual technical rules and adapt the thresholds based on past data. Rules generated by the GP can achieve performances that cannot be achieved by those individual technical rules in their normal usage. EDDIE (which stands for Evolutionary Dynamic Data Investment Evaluator) is a forecasting system to help investors to make the best use of the information available to them. Such information may

include technical rule indicators, individual company's performance indicators, expert predictions, etc. The first implementation, EDDIE-1, was applied to the horse racing domain, whose similarity with financial forecasting has well been documented. The idea was then extended to financial forecasting. EDDIE allows a potential investor to make hypotheses about the factors that are relevant to a forecast. It then tests those hypotheses using historical data and evolves, by way of natural selection, decision trees which aim to provide a good return on investment (ROI). Preliminary but promising results were presented in. FGP (Financial Genetic Programming) is a descendent of EDDIE.

8.5.1 Background

Evolutionary computation has been applied to a broad range of problems with some success from traditional optimization in engineering and operational research to non-traditional areas such as data mining, composition of music and financial prediction. In FGP, a candidate solution is represented by a *genetic decision tree* (GDT). The basic elements of GDTs are *rules* and *forecast values*, which correspond to the *functions* and *terminals* in GP. Figure 8.20 shows an example of a simple GDT. A useful GDT in the real world is almost certainly a lot more sophisticated than this. In GP terms, the questions in the example GDT are *functions*, and the proposed actions are *terminals*, which may also be forecast values. In this example, the GDT is binary; in general, this need not be the case. A GDT can be seen as a set of rules. For example, one of the rules expressed in the GDT in Figure 8.20 is: IF X's price-earning ratio is 10% or more below the average in DJIA AND X's price has risen by 5% or more than the minimum price of last 63 days, THEN Buy X. Improving Technical Analysis Predictions.

GP is attractive for financial applications because it manipulates GDTs (as opposed to strings in GAs). This allows one to handle rule sets of variable size (typical GAs use representations with fixed length). Besides, rules are easy to understand and evaluate by human users, which makes them more attractive than neural networks, most of which are black boxes. For a GP to work, one must be able to evaluate each GDT, and assign to it a *fitness* value, which reflects the quality of the GDT to solve the problem at hand. GP maintains a set of GDTs called a population and works in iterations. In each iteration, FGP creates a new generation of population using standard genetic crossover, mutation and reproduction operators.

Given two GDTs (called parents), the crossover operator in FGP works as follows:

a) randomly select a node within each parent GDTs as a crossover point,
b) exchange the subtrees rooted at the selected nodes to generate two children.

Mutation is employed to keep a population with sufficient diversification. It works as follows:

a) randomly select a node within parent tree as the mutation point
b) generate a new tree with a limited depth
c) replace the subtree rooted at the selected node with the generated tree.

In general, crossover is employed with high probability (e.g. 0.9), whereas mutation has a low probability e.g. 0.01) or less depending on problems. Besides, the reproduction is also used in the process to evolve new generation as asexual operator (e.g. probability 0.1) in order to increase the number of occurrences of individual GDTs with higher fitness. FGP provides the users with two selection strategies, namely *roulette wheel* and *tournament*. Tournament selection is used in this application tests. Its algorithm, which is simple to implement, is described in the following pseudo code:

```
Tournament_Selection()
Begin
Randomly select N individuals from population:
/* N is tournament size */
Choose the one individual with the highest fitness
value among the N individuals
End
```

8.5.2 FGP for Predication in DJIA Index

The Dow Jones Industrial Average (DJIA) index data was taken from 7 April 1969 to 11 October 1976 (1,900 trading days) as training data (or in-sample data) to

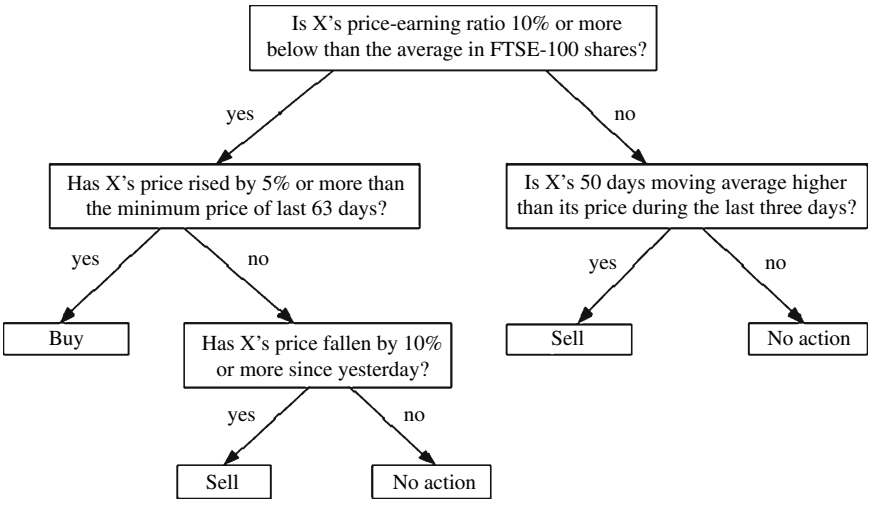

Is X's price-earning ratio 10% or more below than the average in FTSE-100 shares?
yes
no
Has X's price rised by 5% or more than the minimum price of last 63 days?
Is X's 50 days moving average higher than its price during the last three days?
yes
no
yes
no
Buy
Has X's price fallen by 10% or more since yesterday?
Sell
No action
yes
no
Sell
No action

Fig. 8.20 A (simplistic) GDT concerning the actions to take with Share X

generate GDTs, and tested them on the data from 12 October 1976 to 5 May 1980 (900 trading days), which is referred to as "test data". A population size of 1,800, crossover rate of 90%, reproduction rate of 10% and a mutation rate of 1% was used. The termination condition was 2 hours on a Pentium II (200 MHz) or 30 generations. Each run generates one GDT; 20 runs were completed in this experiment. The generated rules were used to predict whether the following goal is achievable at any given day: *Goal G: the index will rise by 4 % or more within the next 63 trading days (3 months)*. Accordingly, each trading day is classified into "buy" category if G holds or "not-buy" category if G does not hold. The whole training and test period contained roughly 50% of "buy" categories. Conclusions could be either *Positive* (meaning that G is redicted to be achievable) or *Negative*. Six technical rule indicators were derived from rules in the finance literature, such as

(1) MV_12 = Today's price – the average price of the previous 12 trading days
(2) MV_50 = Today's price – the average price of the previous 50 trading days
(3) Filter_5 = Today's price – the minimum price of the previous 5 trading days
(4) Filter_63 = Today's price – the minimum price of the previous 63 trading days
(5) TRB_5 = Today's price – the maximum price of the previous 5 trading days (based on the Trading Range Breakout rule [Brock et. al. 1992]).
(6) TRB_50 = Today's price – the maximum price of the previous 50 trading days

Each of the above six indicators is related to some technical analysis rules in the literature. The corresponding six individual technical rules were compared with the GDTs generated by FGP in terms of two criteria: *prediction accuracy* (the percentage of correct predictions) and *annualised rate of return* (ARR). Six individual technical rules using the above six indicators generate "buy" or "not-buy" signals in the following ways. The *moving average rules* (1) and (2) generate "buy" signals if today's price is greater than the average price of the preceding n days ($n = 12$ and 50 respectively). The *filter rules* (3) and (4) generate "buy" signals if today's price has risen 1% than minimum of previous n days ($n = 5$ and 63 respectively). Here 1% is a threshold which an investor has to choose. The *trading range breakout rules* (5) and (6) generate "buy" signals if today's price is greater than the maximum price of previous n days ($n = 5$ and 50 respectively). ARR was calculated based on the following trading behaviour:

Hypothetical Trading behaviour: *It is assumed that whenever a buy signal is indicated by a rule, one unit of money was invested in a portfolio reflecting the DJIA index. If the DJIA index rises by 4% or more at day t within the next 63 days, then the portfolio can be sold at the index price of day t. If not, the portfolio will be sold on the 63rd day, regardless of the price.*

For simplicity, transaction costs and any difference between buying and selling prices are ignored. Rules generated by FGP were tested against the above six individual technical rules in the test data. Results are shown in the column of "On test data I" in Table 8.15. Among the six technical rules, Filter_5 performed best in this set of data. It achieved an accuracy of 52.67% and ARR of 23.03%. The 20 FGP achieved an accuracy of 57.97% in average and an average ARR of 27.79%, which

Table 8.15 Performance comparisons between 20 FGP-generated GDTs and six technical rules on test data I (12/10/1976-05/05/1980-900 trading days) and on test data II (10/04/1981-29/10/1984-900 trading days) of the Dow Jones Industrial Average (DJIA) Index

20	On test data I		On test data II	
	Accuracy	ARR	Accuracy	ARR
Rule 1	60.22%	27.56%	53.89%	55.25%
Rule 2	53.67%	23.57%	53.11%	54.74%
Rule 3	62.00%	31.71%	57.89%	57.80%
Rule 4	58.00%	36.55%	52.33%	61.81%
Rule 5	60.22%	28.23%	63.33%	58.53%
Rule 6	55.11%	29.76%	55.00%	67.00%
Rule 7	61.33%	30.52%	58.00%	56.23%
Rule 8	57.89%	27.16%	54.00%	54.54%
Rule 9	60.67%	28.75%	61.56%	57.69%
Rule 10	55.78%	26.34%	59.11%	59.83%
Rule 11	62.44%	25.93%	55.22%	57.37%
Rule 12	56.78%	25.88%	56.44%	53.19%
Rule 13	56.11%	26.85%	56.44%	59.91%
Rule 14	60.56%	29.66%	54.89%	53.12%
Rule 15	54.78%	25.43%	55.11%	58.05%
Rule 16	56.00%	25.82%	58.11%	59.10%
Rule 17	60.56%	29.18%	58.00%	58.82%
Rule 18	53.00%	23.82%	58.67%	56.61%
Rule 19	60.67%	28.80%	62.89%	58.56%
Rule 20	53.67%	24.18%	57.22%	56.38%
Highest	62.44%	31.71%	63.33%	67.00%
Lowest	53.00%	23.57%	52.33%	53.12%
Mean	**57.97%**	**27.79%**	**57.06%**	**57.73%**
Standard Deviation	3.07%	3.06%	3.06%	3.15%
6 Technical Rules	(the best result in each column is highlighted)			
PMV_12	51.44%	20.68%	44.89%	36.66%
PMV_50	42.56%	16.94%	41.89%	46.85%
TRB_5	49.44%	18.18%	47.44%	55.33%
TRB_50	47.44%	-5.34%	48.67%	**67.00%**
Filter_5	**52.67%**	**23.03%**	**49.44%**	54.53%
Filter_63	50.56%	22.77%	48.89%	48.76%

is better than the Filter_50 rule. In fact, even the poorest GDT achieved an accuracy of 53.00 (rule 18) and ARR of Improving Technical Analysis Predictions Page 8 of 13 23.57% (rule 2), which are still better than the Filter_50 rule. Results show that FGP is capable of generating good rules based on the same indicators used by the technical rules.

Following is the simplest GDT produced by FGP:
(IF (PMV_50 < -28.45) THEN Positive
ELSE (IF ((TRB_5 > -29.38) AND (Filter_63 < 66.24))
THEN Negative

ELSE Positive))

This rule suggests that if today's price is more than 28.45 below the average price of last 50 days, then it is good time to buy (i.e. the rule predicts that one could achieve a return of 4% or more within the next 3 months); otherwise whether one should buy or not depends on the values of TRB_5 and Filter_63. If today's price is no more than 29.38 above the maximum price of the previous 5 trading days or today's price is more than 66.24 above the minimum price in the last 63 days, then it is another good chance to buy. The logical structure and thresholds involved in the above rule may not be found manually by investors. The success of FGP in this set of test data was due to effective combination of individual technical indicators and exploration of thresholds in these rules. To test the robustness of the 20 GDTs across different time periods, the rules were applied to a more recent period and tested them on the DJIA index from 10 April 1981 to 29 October 1984 (900 trading days), which is referred to as "test data II". The test results are illustrated in the column of "On test data II". The GDTs achieved an average accuracy of 85.06%, which out-performs all the six technical rules. As in test data set I, even the poorest GDT performed better than all the technical rules on prediction accuracy. The GDTs achieved an average ARR of 57.73%, which is also better than the ARRs produced by the technical rules except the TRB_50 rule. Two issues are worth pointing. First, although the number of runs is relatively small, the results are significant because the amount of data tested is large and the results are consistent. It is encouraging to see that the GDTs achieve nearly the same mean of accuracy (57.97%, 57.06%) with almost the same value of standard deviation (3.07%, 3.06%) in the two test periods of different characteristics (test period II was more bullish than test period I). Second, it should be pointed out that the calculation of ARR assumes that funds are always available whenever a positive position is predicted and such funds. It achieved the lowest ARR in test data I (−5.34%) but the highest ARR in test data II (67.00%). The erratic performance of the TRB_50 rule is partly due to the fact that it generates very few buy signals. Improving technical Analysis Predictions have no cost when idle. Exactly how one can make use of the predictions by the GDTs is a non-trivial issue.

8.5.3 Concluding Remarks

While technical analysis is widely used as an investment approach among practitioners, it is rarely accepted by academics. It is not the role of the programmer or developer to defend technical analysis here, although the results show that there is some predictability in the DJIA index based on historical data alone. The main objective in this application is to illustrate that FGP, a genetic programming based system, can improve technical rules by taking indicators used in them as input and generate decision trees. For the specific task of predicting whether one can "achieve 4% return with 63 trading days in the DJIA index", FGP reliably generated accurate

GDTs. This involves combining indicators in individual technical rules and finding thresholds in different parts of he decision trees by the way of evolutionary back testing using historical data. The experiment results show that the generated rules can achieve better performances which cannot be obtained by individual rules.

8.6 Genetic Programming within Civil Engineering

This application provides a general background to the topic of genetic programming (GP) that is expanded in subsequent sections to demonstrate how GP can be used in structural engineering. Genetic programming (GP) is a domain independent, problem-solving approach in which computer programs are evolved to find solutions to problems. The solution technique is based on the Darwinian principle of 'survival of the fittest' and is closely related to the field of genetic algorithms (GA). However three important differences exist between GAs and GP:

1. *Structure*: GP usually evolves tree structures while GA's evolve binary or real number strings.
2. *Active vs Passive*: Because GP usually evolves computer programs, the solutions can be executed without post processing i.e. active structures, while GA's typically operate on coded binary strings i.e. passive structures, which require post-processing.
3. *Variable vs fixed length*: In traditional GAs, the length of the binary string is fixed before the solution procedure begins. However a GP parse tree can vary in length throughout the run. Although it is recognized that in more advanced GA work, variable length strings are used. The ability to search the solution space and locate regions that potentially contain optimal solutions for a given problem is one of the fundamental components of most artificial intelligence (AI) systems. There are three primary types of search; the blind search, hill climbing and beam search. GP is classified as a beam search because it maintains a population of solutions that is smaller than all of the available solutions. GP is also usually implemented as a weak search algorithm as it contains no problem specific knowledge, although some research has been directed towards 'strongly typed genetic programming'. However while GP can find regions containing optimal solutions, an additional local search algorithm is normally required to locate the optima. Memetic algorithms can fulfill this role, by combining an evolutionary algorithm with problem specific search algorithm to locate optimal solutions.

8.6.1 Generational Genetic Programming

GP has developed two main approaches to dealing with the issue of its generations; generational and steady-state. In generational GP, there exists well-defined and distinct generations, with each generation being represented by a complete population of individuals. Therefore each new population is created from the older population,

which it then replaces. Steady-state GP does not maintain these discrete generations but continuously evolves the current generation using any available genetic operators. This application uses the generational approach to GP.

8.6.2 Applications of Genetic Programming in Civil Engineering

Procedural programming techniques have been successfully used for the detailed design of structures where the path to the final solution is known in advance. However other design stages lack this predefined approach and thus it becomes necessary to adopt self- learning/organizing computing techniques. This application applies GP at the conceptual design stage, of the design process, with the aim of providing the designer with a series of possible solutions that can be carried forward to the later stages.

8.6.3 Application of Genetic Programming in Structural Engineering

All but one of the published applications in civil engineering to date, utilise a GP system to evolve relationships between variables as suggested by Koza e.g. symbolic regression functions. This section describes how GP can be applied to structural optimisation problems by using the tree structure of GP, to represent a building or structure. In 2000 Soh and Yang published an approach that solved the key issue of how to represent the 'node element diagram' of structural analysis as a 'point-labelled tree' (as used in GP). However because the structure (the phenotype) is now different from the GP tree (the genotype), an additional decoding step must be included in the solution procedure before any fitness evaluation can occur. This step was not previously required when evolving regression functions, as these solutions could be applied directly to the problem. Although this is a departure from traditional GP, by utilising this representation, Soh and Yang demonstrated a system that produced better results when attempting to simultaneously optimise the geometry, sizing and topology of a truss than work using other evolutionary search techniques. It is also important to note that this tree will not degenerate into a linear search as its corresponding structure would be a mechanism. A mechanism is not a viable structure in engineering and thus will be heavily penalised by the fitness function and therefore should not be sustained within the population.

8.6.4 Structural Encoding

It is proposed that the GP tree should compose two types of node: inner nodes which are GP functions representing the cross-sectional areas of the members

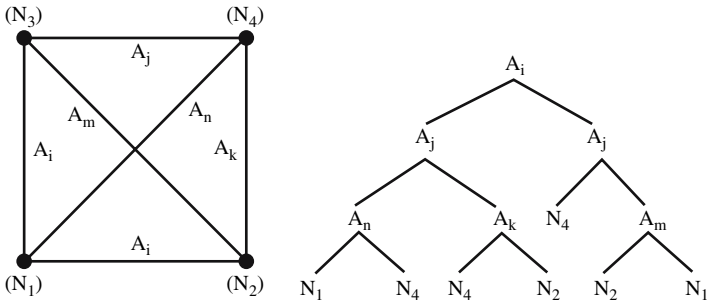


Fig. 8.21 GP tree encoding for structural analysis

A$p(p = \mathrm{i,j,k,l,m,n})$ and outer nodes which are GP terminals representing various node points N$i(i = 1,2,3,4)$ (Figure 8.21). To create a GP parse tree, one member must be selected from the structure to be the root node. This member then has its' corresponding start and end joints represented by children nodes to its left and right. Then from left to right the members associated with each joint node are removed from the structure and used to replace the relevant joint nodes in the GP tree (Figure 8.21). This procedure continues until every structural member is represented in the GP tree.

However it is important that the left-right relationship of child and parent node is maintained as the GP tree is constructed. Therefore each members' start and end joints are represented by the far left and far right children. For example function Aj connects nodes N1 and N2 (Figure 8.21). This approach to structural encoding appears very simple when compared to the complex binary string used by a GA to represent member properties.

8.6.5 An Example of Structural Optimization

This section highlights the differences and improvements between the current work and existing GA and GP based application. The following example aims to minimise the overall weight of the structure by selecting progressively lighter members that can sustain the necessary forces and moments. Whilst it is acknowledged that other factors contribute to the overall cost of a structure e.g material cost and fabrication, structural weight does make a substantial contribution to a structure's cost and therefore it is used here as a first level approximation to a minimum cost structure.

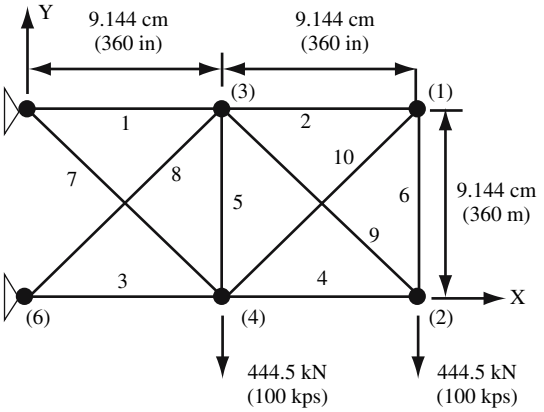


Fig. 8.22 10 member planar truss

8.6.6 10 Member Planar Truss

An illustrative example of a 10-member planar truss is now provided (Figure 8.22) that compares the current GP based application to the results generated by a previous GA and GP based systems. In this example, given the initial structure, the goal was to minimise the structure's weight by varying its' shape and topology. The solution is reported by splitting it into three sections the Model, View and Controller (MVC). These three sections relate to the 'MVC' architecture that this application is based around. MVC separates the underlying GP system from the input GUI and the visualisation output.

8.6.7 Controller-GP Tableau

The Controller allows the user to input data and constraints into the system. With this application, the user can alter the variables that control the evolutionary process, the population size and number of generations. As optimisation problems do not usually have an explicitly defined end point, the GP system runs for a stipulated number of generations before returning the 'best-so-far' individual as the solution. The other essential input for structural optimisation problems is an initial structue to improve. Within this application, the structure is represented by an XML Document Type Definition (DTD) created by MIT called 'StructureXML'. StructureXML not only provides a semantic, storage format for a structure but also provides an interface that allows this system to separate its' structural analysis, evolutionary and visualisation modules. However the StructureXML DTD does not provide any information about a structure's response due to loading. Therefore a new XML DTD was created called 'GPStructureXML' that includes the forces and moments experienced by a structural

Table 8.16 GP Tableau for the 10 member planar truss problem

Objective	For a given 10 Member Planar Truss, subjected to two loads acting simultaneously the objective is to minimise its weight by varying shape and topology.
Terminal Set	Structural joints.
Function Set	Structural members.
Fitness Case	
Raw Fitness	Multiplying the structure's objective function (overall weight) by the structure's corresponding penalty factor.
Standardised Fitness	Equals the raw fitness for this problem.
Hits	
Wrapper	
Parameters	$M = 500,\ G = 51,\ p_c = 0.8,\ p_m = 0.1\ (p_t = 0.04,\ p_o = 0.02)$
Success Predicate	None.

member and the displacement, rotation, reaction forces and moments experienced each individual joint. The following GP Tableau (Table 8.16) summaries the main choices made when applying the major preparatory steps of GP to the problem.

8.6.8 Model

The Model represents the underlying data of the object, in this instance the structural analysis and evolutionary parts of the overall system. As stated previously the each GP tree should be composed of two types of node: the inner nodes which are GP functions representing the trusses members and the outer nodes which are GP terminals representing the structural nodes. This system uses Java based objects to represent each node therefore encapsulation can be used to simplify the genetic operators as the actual data is 'hidden'. When a GP parse tree is created using a fully object orientated language, encapsulation adds to this simple approach. The fitness function rewards a lightweight truss but penalises any structure that violates a specified constraint e.g. maximum allowable member stress. A penalty based approach was employed, rather than outright rejection because the good solutions will typically be located on the boundary between the feasible and infeasible regions.

Therefore this fitness function allows the GP search to approach the optimum solution from the direction of both the feasible and infeasible regions. However, no stochastic search method can guarantee to find the optimal solution but as this system is to be used during the conceptual design stage this is not essential as its' aim is to provide a number of options, to the designer, to be considered and carried forward to the detailed design stage. The structural analysis module is composed of the 'Trussworks' package that allows users to create and analyize 3D structures using the Direct Stiffness Method. Trussworks was selected because its source code is freely available and it is written in the same language as the all other parts of the

system (Java). A finite element package was also considered but this would result in a significant increase in computational power and thus slow down the solution procedure and would also require a StructureXML interface to be created. It was decided that at the conceptual stage a faster but more approximate answer would be advantageous during the design stage. The results from this application indicate that for the 10 bar truss, the GP system produced a weight saving of 0.5%, when compared to a GA based method however it only produced a saving of 0.01%. However the more complicated the structure the greater the potential weight saving of a GP based method over a GA. For example this application found a weight saving of 0.1% when optimising a 25 bar space truss, over the existing GP system and 3.5% over the best GA work. While these improvements may not seem significant it must be remembered that the solution space for these problems is relatively small and that these case studies have been repeatedly used as benchmarks. The solution procedure is shown in Figure 8.23.

8.6.9 View-Visualisation

The View accesses the results data from the Model and specifies how it is displayed. Visualisation is very important at the conceptual design stage as it allows all interested parties e.g. client, architect and engineer to collaborate more closely. Three

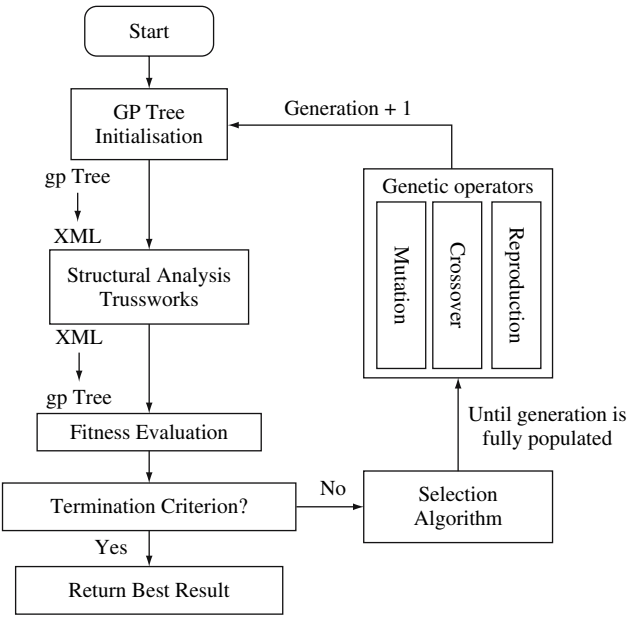


Fig. 8.23 The solution procedure

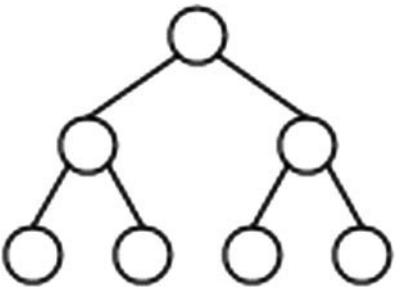


Fig. 8.24 Traditional

types of visualisation are to be employed using this system. The first two relate to the visualisation of the GP parse tree: the traditional and radial while the third displays the actual structure using virtual reality. The traditional approach to viewing GP trees starts with the root node and continues downward, with the terminal nodes at the base of the tree (Figure 8.24). However if the total number of nodes or the tree depth does not remain small then these this method of depiction does not readily allow for the viewer to understand the tree's shape or balance as the representation becomes very large. Radial visualisation solves these problems by mapping the GP tree nodes onto a set of concentric rings according to depth with each node being equally spaced onto the perimeter (Figure 8.25).

The third visualisation transforms any GPStructureXML document produced by the system into a X3D based, virtual reality model. The X3D specification has been developed by the Web3D consortium and is currently under ISO Standard review. Touted as 'next-generation open standard for 3D on the WWW', X3D essentially extends and upgrades the capabilities of VRML97 using XML and allows the user to not only visualize the structure created but also to 'fly' around it to view it from any angle. X3D's extrusion node has been utilized in this visualization process because given a cross-section and length, X3D can produce an object of any size. The main advantage of X3D as a file format is that the visualizations produced are very small

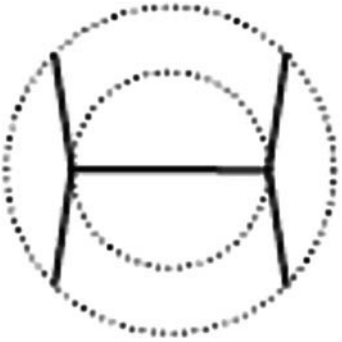


Fig. 8.25 Radial

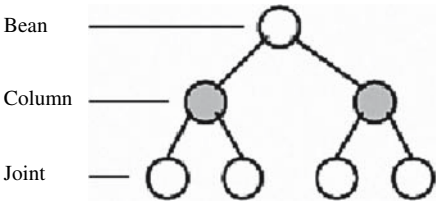


Fig. 8.26 Column/beam binary tree

in size typically under 100kB and thus can easily be transmitted over the internet with ease and secondly they do not require much processing power to view as they can use any Internet browser with the correct 'plug-in' installed. This application aims to extend GP to the conceptual stage of multi-storey office building design. Here the GP will be used to generate several novel solutions some of which the designers will take forward into subsequent design stages. This will provide many challenges as initially any GP parse tree must contain two types of member node i.e. column and beam but both of these members allow the GP tree to remain as a binary tree (Figure 8.26). However if a slab is also to be included because a slab must span between more than two nodes, the GP tree is no longer a binary tree (Figure 8.27).

8.6.10 Concluding Remarks

This GP based system has been shown to provide better results to an optimization problem when compared to an equivalent GA system while using a method of representation that is significantly more simplistic than that used by the GA. The use of XML not only provides an interface between the components of this system and other programs but also allows for the visualisation of the complete structure using a virtual reality model.

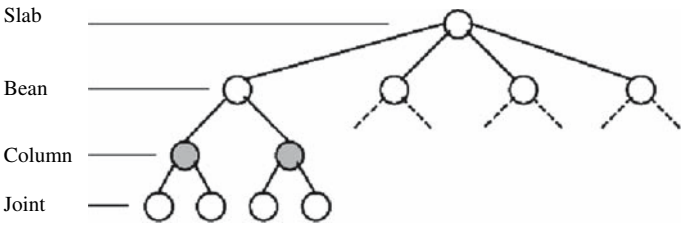


Fig. 8.27 Slab/column/beam binary tree

8.7 Chemical Process Controller Design using Genetic Programming

Most common chemical process control systems can be classified as either reference control systems or regulatory control systems. In simple reference control systems the objective is to get the process output to follow a reference signal (or setpoint) as closely as possible, preferably without using excessive control effort. In the case of regulatory control systems the objective is usually to hold the process output at a desired value in the presence of disturbances. Often the same controller can be used to accomplish both tasks for a given process. Given the above objectives there are, broadly speaking, two ways to design an appropriate control system. The most frequently used method is to pre-select a controller structure and then to tune the parameters of this controller so that the desired closed-loop response is obtained. This is referred to as a parameter optimized control system, the most well known example of which is probably the Proportional Integral Derivative (PID) controller. The other approach is the use of structure optimal control systems, where both the structure and parameters of the controller are adapted to those of the process model. In practice, however, the use of the latter method is severely restricted because exact dynamic term cancellation is required in order to produce the optimal controller structure. This is usually not possible for various reasons, e.g. of the lack of an appropriate process model, non-linearities and physical constraints on the process variables. Genetic programming offers an interesting new perspective on structure optimized controller design because of its potential ability to match controller algorithms to the process model by implicitly utilizing the information about constraints and non-linearities that the closed-loop simulation provides. GP is also an attractive proposition for control system design in that the performance criterion used to perform the optimization does not have to be continuous or differentiable. This means that, potentially, the designer can introduce more 'real world' performance measures to be used directly by the GP algorithm. Recently there have been a number of articles describing the use of GP in designing control algorithms of various types.

8.7.1 Dynamic Reference Control Problem

In this application, GP is used to design recursive discrete-time reference control systems. In both case studies the objective of the GP derived algorithm is to ensure that the process output follows the set point in a desirable manner through step changes of differing magnitudes across a pre-defined operating region. Figure 8.28 shows the type of control configuration used in both process case studies.

A controller that is used in many real chemical process control problems of the type shown in Figure 8.28 is the PID controller, described in its original continuous time form by Equation 8.1.

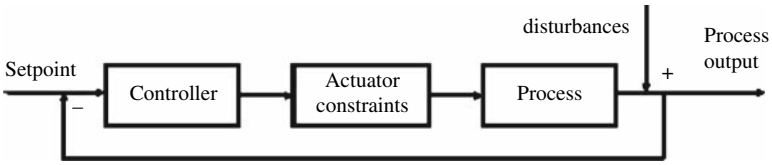


Fig. 8.28 Control configuration used in GP design

$$u(t) = K\left[e(t) + \frac{1}{T_i}\int_0^t e(\tau)d\tau + T_d\frac{de(t)}{dt}\right] + u_{ss} \tag{8.1}$$

Where $u(t)$ is the controller output at time t, $e(t)$ is the setpoint error at time t and *uss* is the controller output at some reference steady state. The PID controller is parameterised by the proportional gain K, the integral time T_i and the derivative time T_d. These are adjusted (tuned) in order to give the desired closed loop response. The $K_e(t)$ term produces a controller output proportional to the size of the error but use of proportional control alone produces tracking offset when applied to most processes i.e. those without natural integrating properties. To counter this, the integral term in Equation 8.1 is used to automatically 'reset' the offset. Additionally, the derivative term may be used to provide anticipatory control by introducing a control effort proportional to the rate of change of setpoint error. This term becomes dominant when the setpoint changes suddenly, e.g. in a step change, and so provides a faster initial response. However, many designers choose to set the derivative term to zero as it has an amplifying effect on noisy measurements. With the increasing use of digital computers for chemical process control the discrete form of the algorithm is now prevalent. For small sample times, T_s, the discretised version of the PID algorithm may be obtained by applying rectangular integration to Equation 8.1 to yield Equation 8.2.

$$u(k) = K\left[e(k) + \frac{T_s}{T_i}\sum_{i=0}^{k-1} e(i) + \frac{T_d}{T_s}(e(k) - e(k-1))\right] + u_{ss} \tag{8.2}$$

The control signal $u(k)$ is now calculated every sample time and, in the case of a continuous process, a zero-order hold element is usually used to hold the control signal steady between sampling times. The recursive form of the algorithm, Equation 8.3, is often preferred because it is computationally simpler and because no special provision need be made for smooth switching from manual mode (open loop) to automatic mode (closed loop). Equation 8.3 can be derived from Equation 8.2 by the use of backward differencing.

$$u(k) = u(k-1) + q_o e(k) + q_1 e(k-1) + q_2 e(k-2) \tag{8.3}$$

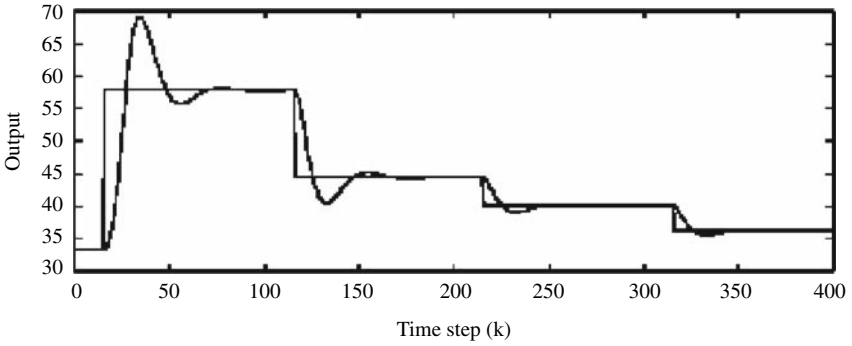


Fig. 8.29 Example of PID reference control

Where,

$$q_o = K\left[1 + \frac{T_d}{T_s}\right] \tag{8.4}$$

$$q_1 = -K\left(1 + 2\frac{T_d}{T_s} - \frac{T_s}{T_i}\right) \tag{8.5}$$

$$q_2 = K\frac{T_d}{T_s} \tag{8.6}$$

A typical PID closed loop response to a series of step changes in setpoint is shown in Figure 8.29. The response shown is for illustrative purposes and is not considered to be well-tuned.

This example is to illustrate that when tuning a PID controller one must trade-off the various transient response properties, e.g. settling time, rise time, overshoot, and degree of oscillation. The desired properties of the response depend largely on the preferences of the designer and the problem context so one cannot easily specify a generic best response that holds over all processes. For example, the designer of a reboiler control system for an industrial distillation column would also want to consider the cost of the control effort (i.e. steam usage) in deciding the best parameter values. In cases such as these, aggregate cost functions containing terms pertaining to actuator effort, and a performance measure such as IAE (Integral of the Absolute Error) or ITAE (Integral of the Time weighted Absolute error) can be utilised, and the parameters can then be optimised using numerical procedures. However, any performance criterion that reflects the designer's *a priori* qualitative description of a 'good' closed-loop response may be used.

8.7.2 ARX Process Description

The ARX process used is of the linear form shown in Equation 8.7.

$$Ay(k) = Bu(k-1) + C\varepsilon\,(k) \tag{8.7}$$

Where $y(k)$ and $u(k-1)$ are the process output at time k and the controlled input at time $k-1$ respectively. *A, B* and *C* are polynomials in the backshift operator $z-1$ and $\varepsilon\ (k)$ is a stationary white noise sequence. The polynomial coefficients used (in order of ascending powers of $z-1$) were $A = \{1 - 0.99\ 0.05\},\ B = \{0.1\ 0.01\}$ For the purposes of this study *C* was set to zero to yield an entirely deterministic system. In practice, the response of the output sequence $y(k)$ to $u(k)$ can most easily be calculated by rearranging Equation 8.7 and after substitution of the appropriate coefficients, proceeding in the recursive manner described by Equation 8.8.

$$y(k) = 0.99y(k-1) - 0.05y(k-2) + 0.1u(k-1) + 0.01u(k-2) \tag{8.8}$$

Additionally, when this simulating this process in the configuration shown in Figure 8.30 the following constraints were imposed on the system:

- Setpoint changes: constrained between limits of 60 units (high) and 20 units (low). Process output also limited between these constraints.
- Process Input $u(k)$: constrained between hard limits of 90 units (high) and 0 units (low) and a rate-limit of a maximum of magnitude 4 units change per discrete time step.

8.7.3 CSTR (Continuous Stirred Tank Reactor) Process Description

The simulated continuous stirred tank reactor system used in the second case study is shown in Figure 8.30. A feed stream that has a high concentration of reactant

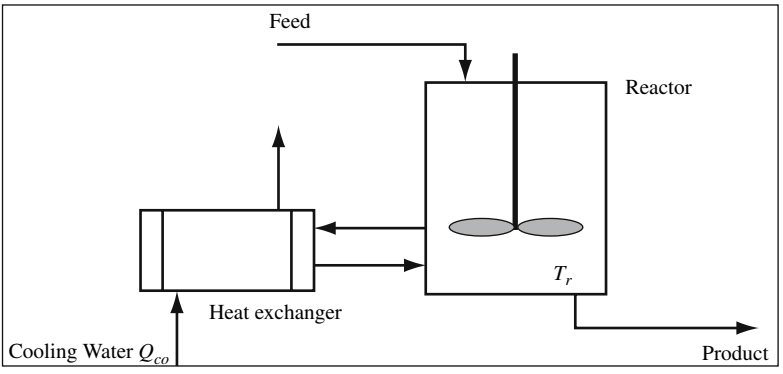


Fig. 8.30 Schematic of CSTR and heat exchange system

A enters the reactor. Then, within the reactor, the following exothermic reaction takes place: **A** → **B** → **C**. Reactant **C** is regarded as the undesired byproduct of the reaction and at high temperatures most of reactant **B** is converted to **C**. The reactor is cooled by means of an external pump-around heat exchanger. The reactor product stream temperature, T_r, and hence the concentration of the various products within it, is controlled by manipulating the flow rate of cooling water, Q_{CO}, to the heat exchanger.

The following state variables are defined: T_r is the temperature of the reactor product stream, T_{RC} is the temperature of the reactor product recycle stream and T_{CO} is the temperature of the coolant stream on leaving the heat exchanger. C_A and C_B are the concentrations of Reactant A and Reactant B in the reactor product stream. Equations 8.9 and 8.10 are material balances on reactants A and B respectively. Equation 8.11 describes the thermal energy balance over the reactor. Equation 8.12 describes the energy balance for the reactor product recycle stream and Equation 8.13 describes the energy balance for the coolant passing through the heat exchange unit.

$$V_r \frac{dC_A}{dt} = -K_A e^{\frac{-E_A}{RT_r}} C_A V_r + Q_F(C_{AF} - C_A) \tag{8.9}$$

$$V_r \frac{dC_B}{dt} = K_A e^{\frac{-E_A}{RT_r}} C_A V_r - K_S e^{\frac{-E_B}{RT_r}} C_S V_r + Q_F(C_{BF} - C_A) \tag{8.10}$$

$$\rho c_p V_r \frac{dT_r}{dt} = \Delta H_A K_A e^{\frac{-E_A}{RT_r}} C_A V_r + \Delta H_B K_B e^{\frac{-E_B}{RT_r}} C_A V_r + \rho c_p Q_F(T_F - T_r) + \rho c_p Q_{RC}(T_{RC} - T_r) \tag{8.11}$$

$$\rho c_p V_{RC} \frac{dT_{RC}}{dt} = \rho c_p Q_{RC}(T_r - T_{RC}) + U_H A_H(T_{CO} - T_{RC}) \tag{8.12}$$

$$\rho c_p V_{CL} \frac{dT_{CO}}{dt} = \rho c_p Q_{CO}(T_{CIN} - T_{CO}) + U_H A_H(T_{RC} - T_{CO}) \tag{8.13}$$

Additionally, the following constraints were imposed on the system:

- Reactor temperature, Tr : constrained to follow set points generated between limits of 373 K and 325 K.
- Cooling water flow rate, Q_{CO}: constrained between hard limits of 0 litres/sec and 10 litres/sec. A rate limit of a maximum cooling water magnitude change of 5 litres/sec per 5-second interval (i.e. every discrete time step) was also imposed. The physical parameter values for this system are shown in Table 8.17 and were fixed for all the simulations carried out.

The CSTR system equations were simulated using a Runge-Kutta 4th order method with a fixed step size of 5 seconds, both the integration method and step length were determined in a trial and error manner.

Table 8.17 Model parameters for CSTR system

Symbol	Description	Unit	Value
V_r	Reactor Volume	m^3	3
K_A	Rate Coefficient A $\rightarrow$ B	mol/sec	7×10^{11}
K_B	Rate Coefficient B $\rightarrow$ C	mol/sec	9×10^{11}
E_A	Activation Energy A $\rightarrow$ B	J	90000
E_B	Activation Energy B $\rightarrow$ C	J	100000
R	Gas Constant	J/mol K	8.314
Q_F	Reactor feed flowrate	m^3/sec	0.015
Q_{RC}	Heat exchange recycle flowrate	m^3/sec	0.025
C_{AF}	Conc. of reactant A in feed stream	mol/m^3	2500
C_{BF}	Conc. of reactant B in feed stream	mol/m^3	0
P	Process fluid density	Kg/m^3	1000
C_p	Process fluid heat capacity	J/kg K	4140
U_H	CSTR heat transfer Coefficient	W/m^2K	3000
A_H	Heat exchange area	m^2	100
ΔH_A	Heat of reaction A $\rightarrow$ B	J/mol	80000
ΔH_B	Heat of reaction B $\rightarrow$ C	J/mol	80000
V_{RC}	Heat exchanger process fluid volume	m^3	0.2
T_{CIN}	Cooling water entry temperature	K	293
T_F	Reactor feed temperature	K	320
V_{CL}	Heat exchanger cooling fluid volume	m^3	0.2

8.7.4 GP Problem Formulation

The GP algorithm, for both case studies, is set up to construct a discrete recursive feedback control law of the following form: $u(k) = u(k-1)+$ [output of individual tree]For a population size of M the output of the ith $\{i = 1,2,3...M\}$ controller at time k is equal to the output at time step $k-1$ plus some correction term applied by the ith GP individual. Hence, the controller can be regarded to have a state that is updated by the GP individual (based on current information about the process supplied by an appropriately designed terminal set). The controller output is only limited by the physical constraints peculiar to each process and need not take an on/off only value. The recursive formulation of the controller was chosen because it simplifies computer simulation of the closed-loop systems and has several other properties that make it a more practical implementation than the non-recursive form.

Fitness Function

The fitness of an individual, F_i, is constructed as follows:

$$F_i = \sum_{j=1}^{p} \frac{\sum_{k=0}^{n} k\,|e(k)| + r\Delta u^2(k)}{\Delta S_j} \tag{8.14}$$

This is calculated by simulating the closed loop behaviour of the ith controller over P independent and randomly generated setpoint changes: $j\Delta S\,\{j = 1, 2, 3 \ldots P\}$ Each closed-loop step change response was evaluated from 0 to n discrete timesteps. The value of n was determined before the GP run by a trial and error procedure to give sufficient time for a reasonable controller to settle to its steady state value. The performance criterion, F_i, contains an ITAE term that is calculated with respect to some 'desirable' reference trajectory (in both cases this was set to equal the setpoint trajectory delayed by one discrete time step). The ITAE weighting factor for the sum of absolute errors was set to be simply the value of the current timestep k. The performance criterion also contains a weighted (with a constant r) penalty term for excessive control effort. In practice this was determined, again, in a trial and error manner before the GP run. The inner summation term is then scaled according to the size of the step change $j\Delta S$ in order to ensure that contributions from 'small' setpoint changes are not dominated by those from 'large' changes.

Randomization of Training Cases

In each case study the setpoint trajectories were generated randomly from generation to generation, thus giving rise to a non-stationary fitness function. The rationale behind this was the attempt to produce controllers that generalize well over the operating region of the process. Some preliminary runs using the GP with the same number of fixed setpoint trajectories throughout the entire run almost always led to very brittle solutions, i.e. controllers that performed badly on step changes they were not explicitly trained upon.

This effect appeared to be more deleterious in the case of the CSTR than the ARX system. A possible explanation for this is that the CSTR's inherent non-linearities means that the limited subset of training cases utilized in the GP run are less likely to accurately encapsulate the dynamic behavior across the entire operating region of the process. This leads to poor generalization properties when the GP is trained on small fixed training set of step changes.

8.7.5 GP Configuration and Implementation Aspects

Both the GP and the process simulations were implemented within the Matlab programming environment and runs were performed on a desktop PC. The GP algorithm is of a standard nature, using the operators of direct reproduction, mutation and crossover. Table 8.18 summarizes the configuration and algorithm control parameters used in both case studies. They were decided upon on the basis of available computing time and the results of some preliminary runs. Selection using linear ranking was chosen to avoid problems with fitness scaling, which would be difficult to implement due to the non stationary fitness measure. The functional set is fairly standard except for the 'backshift operator' designated as the Z operator in

Table 8.18 GP run configuration

Population size M	200
Termination criterion	100 generations elapsed
Fitness cases:	4–8 step-changes generated randomly each generation.
Fitness measure	The performance measure F_i as defined in Section 5.1
Number of discrete steps n	ARX:100 CSTR: 140
Selection method	Linear ranking.
P (Crossover)	0.8
P (Direct reproduction)	0.1
P (Mutation)	0.1
Elite individuals copied to next generation.	10%
Max program size	Approx. 100 nodes.
Terminal set	Current and 2 past values of setpoint error *e(k)*. Current and 2 past values of process output *y(k)*. Random constants in range: $[-100 \rightarrow 100]$.
Functional set	$+ - \%$ * `EXP PLOG PSQRT IFLTE Z NLTF` (see below)
GP runs per case study	25
Wrapper	Case specific actuator constraints.

this application and the variable non-linearity sigmoid, which is designated NLTF (Non-Linear Transfer Function). The backshift operator has parity 1 and outputs the value of the input sub-tree evaluated at the previous time-step. This operator, when nested, has the potential to select terminals and sub-trees from several time steps into the past and so any controllers created need not rely on just the past 2 values of the process output and the setpoint error as stated in Table 8.18. The variable sigmoid NLTF has parity 2 and implements the function described by:

$$NLTF(x,y) = \frac{1 - e^{(-xy)}}{1 + e^{(-xy)}}$$

Where the values of x and y at any given time step determine the non-linearity of the sigmoid, which is similar in properties to the basis functions used in many Artificial Neural Networks. The terminal set consists of those terms most likely to be correlated with controller performance i.e. past and present values of errors and process output. Past values of controller output were not included because a number of preliminary runs using them failed to yield any controllers that could produce an acceptable response over a validation sequence of step changes.

8.7.6 Results

After the GP runs were performed each best of run individual (BRI) was tested against a predefined fixed validation sequence of setpoint step changes. This enabled controllers from different runs to be compared in a qualitative manner (i.e. the shape of the responses.) In order to provide a general yardstick for the behaviour of the controllers the validation sequence responses were also compared with those generated by recursive-form PID controllers. These were manually tuned to give a fast response with a low overshoot, and a Simplex optimization was performed to fine-tune them further. The manually obtained PID parameters were used as starting values and *Fi* (over the validation sequence) as the objective function. In addition to the validation step sequence tests it was decided to observe the behaviour of the evolved controllers over a range of conditions that were not present during the training phase, i.e. to see how well they generalised. Two different tests were carried out on each BRI: (1) Setpoint tracking with respect to a differently shaped reference signal (in this case a ramp signal.) and measurement noise on the output. (2) The rejection of a series of load disturbances applied at the process output.

ARX System Results

Of the 25 runs performed, several produced BRIs that could be simplified to linear recursive form PI or PID controllers, but most produced controllers that were not of the PID type and often included the non-linear functional such as NLTF. Additionally, when tested over the validation step change sequence, nearly half the runs produced BRIs that were deemed unacceptable due to anomalous behavior; e.g. they had large offsets or very oscillatory responses. Figure 8.31 shows a typical

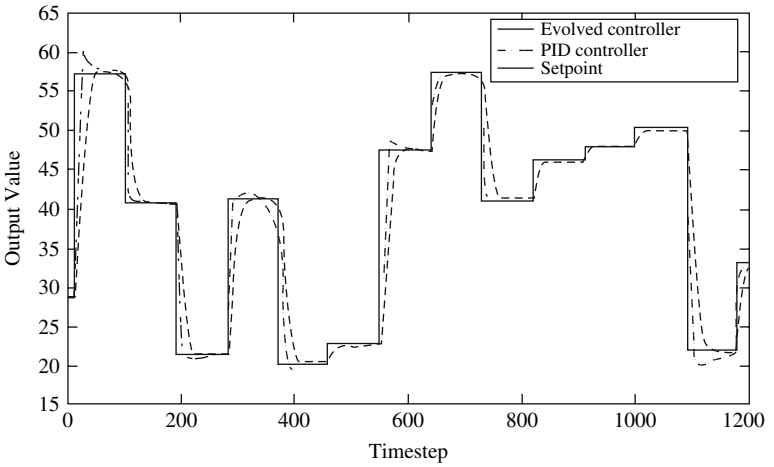


Fig. 8.31 ARX: nominal process response

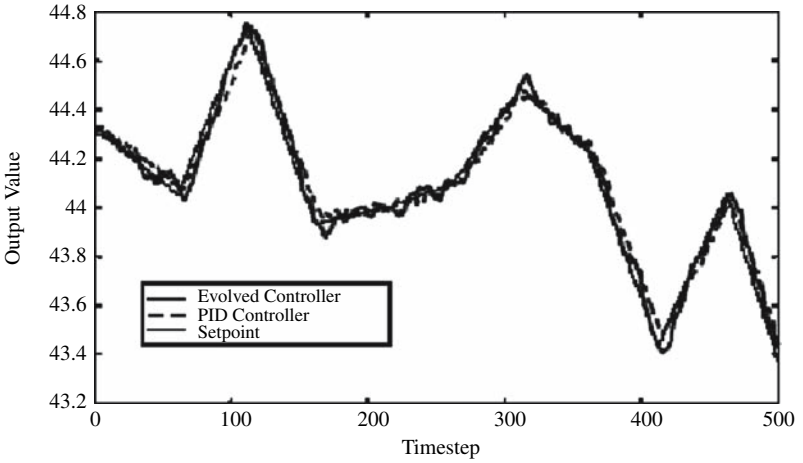


Fig. 8.32 ARX: ramp reference input with noise

closed-loop response of the ARX process using a non-linear BRI and the nominal model. The response of the PID controller is also shown for comparison. Qualitatively, it can be seen from the responses that the evolved controller offers similar performance to that of the PID controller. In the case of a ramped reference signal and noise most of the evolved non-linear controllers generalized well and gave PID levels of performance (Figure 8.32.)

In the disturbance rejection tests, however, it became apparent that the evolved controllers' robustness did not extend to classes of control objectives not explicitly described during the training phase. All of the evolved ARX controllers (except the linear ones) became unstable or performed badly at rejecting output load disturbances (Figure 8.33.)

CSTR System Results

The proportion of runs that generated acceptable controllers was similar to that obtained for the ARX system and again a proportion of the evolved controllers could be reduced to the form of linear controllers. Figures 8.34 and 8.35 show the response of the nominal CSTR system with a non-linear evolved controller (with a PID controller for comparison) and the corresponding cooling water flow rates. It can be seen that although the controllers have very similar performance, the evolved controller uses less control effort. On generalization testing, the BRIs exhibited no real difference to the ARX case, and behaved in a very similar manner on the ramp tracking problems. The non-linear controllers evolved for the CSTR also tended to go unstable, or offer a very poor response, when presented with the regulatory problem of load disturbances on the output, although due to space limitations, this is not illustrated.

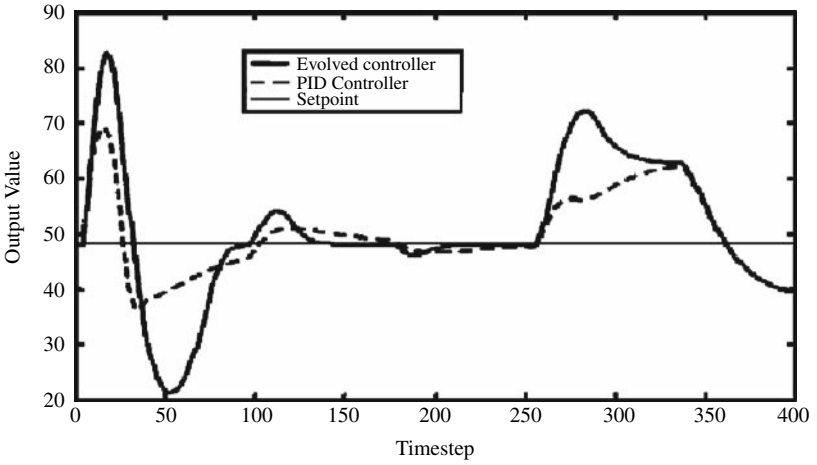


Fig. 8.33 ARX: Load disturbances on output

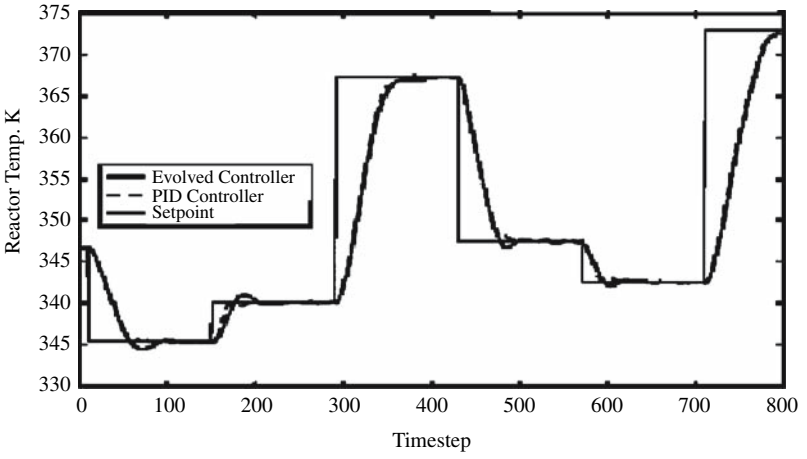


Fig. 8.34 CSTR: nominal process response

8.7.7 Concluding Remarks

It has been shown that the GP framework is capable of producing dynamic recursive controllers for two example simulated processes for a particular class of control problem. It was found that the control algorithms seemed to be able to generalize across the process operating regions without any difficulty when a random fitness case based scheme was used during training. However, as expected, these control algorithms are brittle in the face of some situations they have not encountered e.g. disturbance rejection problems. A major problem is that of controller stability, even for the nominal case there does not appear to be any simple way of incorporating a

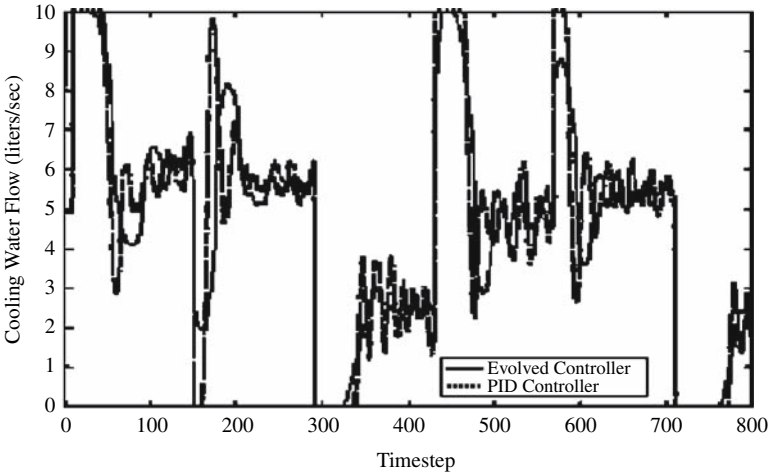


Fig. 8.35 CSTR: manipulated flowrate

stability criterion into the GP procedure, or even establishing stability of the controllers *aposteori*. These issues will have to be addressed if GP is ever going to be used for real controller applications. The fact that GP can offer similar performance to the PID controllers in these cases is encouraging, however. Future work will concentrate on the robustness issues presented by GP derived control laws (e.g. can controller robustness be improved by a more intelligent scheme for presenting the training cases?) Additionally, the technique will be applied to more challenging control problems, i.e. those in which traditionally designed controllers fail to operate effectively.

8.8 Trading Applications of Genetic Programming

At Algometrics, novel artificial intelligence [AI] and advanced statistical methods are applied to develop trading strategies for a wide range of securities and derivatives. As part of this application some powerful tools such as genetic programming ("GP") is used to solve problems that are mathematically intractable or where there is no clear solution domain dictated by the nature of the problem.

Experimentation with very powerful but brute force methods like GP can lead to the development of new insights or promising partial solutions that can be investigated with other more focussed techniques. GP on its own is unsuitable for all but fairly trivial problems because the potential search space is often huge and the data for most time series is relatively short, leading to data mining, or in the case of GP, model mining biases. The search space can be constrained or reduced by either selecting particular problems that are amenable to short parsimonious solutions or

constraining the search space through priors relating to what may already be known about the solution to the problem.

A large part of successfully applying powerful AI techniques to financial problems, therefore, lies in the selection of the appropriate question rather than the unguided application of an automated answer generation procedure. Finding sensible questions to ask is often a far more difficult problem than the operationally efficient implementation of a technique such as GP.

In the case of developing trading systems, using the most appropriate techniques for the question to hand is critically important. The reason is that model induced risk can be considerable unlike say, engineering where for most problems there is well tested theory that can give us considerable comfort that the solution found by a method like GP is a sensible one. In finance, therefore, it is important to exercise care in the blind application of powerful search or optimization methods.

This problem is solved by using GP to identify new ideas that may be promising for further investigation and refinement. As such, GP can be viewed as a kind of generic methodology that can be run automatically with the potential of finding a market anomaly, group of securities or trading method that can either enhance an existing strategy or open up a larger area of productive research.

There is also the more prosaic point that if you have any spare computing capacity lying around it makes sense to use it even for fairly unproductive data mining exercises. If it is otherwise idle, the marginal increase in productivity is infinite.

Genetic programming usually proceeds by the action of a set of genetic operators such as crossover and random mutation on a suitable encoding representation of a population of LISP parse tree structures representing the particular trial solutions (programs). An iteration of the search algorithm is defined by the truncation of the modified set of trial solutions based on fitness or objective function, often penalized for the model encoding length, and subsequent propagation of the better solutions using a variety of probabilistic balancing rules.

After a large number of such iterations the population will come to be dominated by viable solutions and the general hope is that this represents the global optimum and can be reached in a reasonable time scale. It is a hope because the theoretical convergence properties of GP are not well understood and the global optimum may never be reached.

This approach differs from the conventional one in that instead of using LISP parse tree structures as the system representation, fully connected neural networks are used. These networks then undergo node pruning and connection modifications using an objective function that is heavily penalized by model complexity. Minimal models are more likely to represent correct solutions rather than an artifact of data over-fitting.

This procedure requires a double optimization. A search over the space of possible candidate models using GP and selection of the parameters for each of these models. It turns out that for many time series problems this reduces the search time considerably and can make use of fast and efficient methods of training neural networks.

The search algorithm also differs slightly from convention in that a hybrid genetic algorithm ("GA") is used, even if slower than some implementations of GAs benefits from known convergence properties for very large classes of problem. The model representation chosen is also more suited to real valued data and tends to be fairly stable with respect to pathological data points. For a recurrent network there is clearly a stability issue and an important mathematical trick is to constrain the network parameters in such a way that convergence is guaranteed.

GP applications can be divided into three distinct categories. The first is the identification of determinism in financial time series by setting up the forecasting or prediction problem in a fairly conventional way. The second application looks for minimal causal relationships and links in a range of data types that varies in terms of its sampling frequency (quarterly through to tick) and often across quite unrelated time series (for example, temperature and climate data is extremely useful in trading soft commodities). Finally, given a set of expected returns simulated data and GP techniques are used to build ad-hoc trading rules and portfolio rules that are simple and very easy to implement in practice.

8.8.1 Application: Forecasting or Prediction

The standard prediction problem is by far the easiest to set up and a thoroughly explored application area in the neural network literature. Here lagged values of one or more time series are used to map to an expected future value a given number of time steps away via a mapping function defined by the neural network. There are a couple of wrinkles in this implementation that are unusual. First, the mapping can be via a recurrent neural network rather than the feed forward perceptron usually encountered in the literature. This is advantageous because it is better at handling purely memory associative mappings (in terms of parsimony) rather than a mapping defined by a smooth function. Second, and more important, the time steps can be variable, in fact a function of the data set. This is important in financial time series analysis because the arrival of large trades that have a serious affect on prices can be irregularly spaced and can result in persistence. GP is used to generate simple networks that are trained on a suitable objective function such as a prediction error and promising classes of network are further explored, the end result often being a very simple model based on a generalization of the results obtained from the GP step. The important point is that automation of the search and the ability to conduct a very large number of tests often provides an illuminating insight into the important features of the problem. For example, the number of lags that are required, the non-linearity of the best mapping, time series decimation rules and optimal model complexity.

8.8.2 Application: Finding Causal Relationships

There is a vast amount of financial data available including price data, earnings and corporate data, economic statistics and other statistical data relating to particular markets, products, commodities or goods prices. Traders are always seeking explanatory or forecasting ability between one time series and another. GP methods are used to find causal links and minimal explanatory models between time series that can be linked by potentially complex and non-linear relationships.

Standard factor model methods such as principal components analysis, whilst insightful, are not able to generate minimal models and building non-linear factor models without constraining priors on the relevant components is a computationally intractable problem given the size of the available data set. The objective here is to use GP methods to explore links and relationships in a quasi-automatic way and report interesting or promising relationships. For example, the factors that might affect a commodity price or interest rate. GP is not used to actually build a model of the appropriate relationships but leave this to other statistical methods. The important advantage of using GP in this way is again to illuminate what might be an appropriate relationship not to generate an optimal model of that relationship.

8.8.3 Application: Building Trading Rules

One application where GP is used directly is in the generation of trading rules and sub-components of trading systems. These rules typically relate to generic portfolio construction methodology and are conditioned on a real data set. Given a trading signal the objective is to find portfolio construction or trade implementation rules that optimize an objective function such as a risk adjusted expected return.

So for example, suppose the idea is to find a set of rules conditioned on short-term tick data that minimizes leg risk in executing an index arbitrage trade. For this a measure of commonality between the index is found. The idea in this case is to use this indicator (a function of the number of stocks, liquidity and covariance with the index future) and say the best bid or offer or size to work into the position maintaining maximum tracking, minimizing cost and market impact.

This is a non-trivial problem when set up correctly to solve using a GP technique. The advantage of this sort of exercise is that the problem is well defined and it may sometimes be possible to use simulated data rather than real data. There are many other trading and portfolio problems that can be solved in a similar way, for example trading stock pairs, performance targets, stock concentration rules, optimal trading price limits and rules for executing contingent trades.

8.8.4 Concluding Remarks

GP is a useful tool in financial trading applications when properly applied. The trick, as with any application for a mathematical or computing technique, is to carefully define exactly what the user wishes to achieve and apply the technique only if it is appropriate. Part of the skepticism from finance practitioners for very general optimization techniques using limited data sets arises because the methods have been inappropriately applied not because they are of no value.

Here three problem areas were demonstrated with advances using GP with very little expenditure of research time. All the results were derived in a quasi-automatic way with little human intervention and whilst GP will not be able to generate low cost source code for an arbitrary problem in the foreseeable future it can help us to derive simple rules that are not that obvious.

8.9 Artificial Neural Network Development by Means of Genetic Programming with Graph Codification

NNs (Neural Networks) are learning systems that have solved a large amount of complex problems related to different areas (classification, clustering, regression, etc.). The interesting characteristics of this powerful technique have induced its use by researchers in different environments. Nevertheless, the use of ANNs (Artificial Neural Networks) has some problems, mainly related to their development process. This process can be divided into two parts: architecture development and training and validation. As the network architecture is problem dependant, the design process of this architecture used to be manually performed, meaning that the expert had to design different architectures and train them until he finds the one that achieves best results after the training process. The manual nature of this process determines its slow performance although the recent use of ANNs development techniques have contributed to achieve a more automatic procedure.

8.9.1 State of the Art

Genetic Programming

Genetic Programming (GP) is based on the evolution of a given population. In this population, every individual represents a solution for a problem that is intended to be solved. The evolution is achieved by means of selection of the best individuals – although the worst ones also have a little chance of being selected – and their mutual combination for creating new solutions. This process is developed using selection, crossover and mutation operators. After several generations, it is expected that the

population might contain some good solutions for the problem. The GP encoding for the solutions is tree-shaped, so the user must specify which are the terminals (leaves of the tree) and the functions (nodes with children) for being used by the evolutionary algorithm in order to build complex expressions The wide application of GP to various environments and its consequent success are due to its capability for being adapted to different environments. Although the main and more direct application is the generation of mathematical expressions, GP has been also used in others fields such as knowledge extraction, rule generation, filter design, etc.

ANN Development with EC Tools

The development of ANNs is a topic that has been extensively dealt with very diverse techniques. GA and GP techniques follow the general strategy of an evolutionary algorithm: an initial population consisting of different genotypes, each one of them codifying ANN parameters (typically, the weight of the connections and/or the architecture of the network and/or the learning rules), is randomly created. This population is evaluated in order to determine the fitness of each individual. Afterwards, this group is made evolve repeatedly by means of different genetic operators (replication, crossover, mutation, etc.) until a determined termination condition is fulfilled. This condition can be, for example, if a sufficiently good individual is obtained, or that a predetermined maximum number of generations is reached. As a general rule, the field of ANN generation using evolutionary algorithms is divided into three main fields: evolution of weights, architectures and learning rules. First, the weight evolution starts from an ANN with an already determined topology. In this case, the problem to be solved is the training of the connection weights, attempting to minimize the network failure. Most of the training algorithms, such as backpropagation (BP) algorithm, are based on gradient descent, which presents several inconveniences, mainly the possibility of getting stuck into a local minimum of the fitness function. With the use of an evolutionary algorithm, the weights can be represented either as the concatenation of binary values or real numbers. The main disadvantage of this type of encoding is the permutation problem. This problem means that the order in which weights are taken at the array can make equivalent networks correspond to completely different chromosomes, making the crossover operator inefficient. Second, the evolution of architectures involves the generation of the topological structure. This means establishing the connectivity and the transfer function of each neuron. The network architecture is highly important for the successful application of the ANN, since the architecture has a very significant impact on the processing capability of the network. Therefore, the network design, traditionally performed by a human expert using trial and error techniques on a group of different architectures, is crucial. The automatic architecture design has been possible thanks to the use of evolutionary algorithms. In order to use them to develop ANN architectures, it is needed to choose how to encode the genotype of a given network for it to be used by the genetic operators. At the first option, direct encoding, there is a one-to-one correspondence between every one of the genes and their

subsequent phenotypes. The most typical encoding method consists of a matrix that represents an architecture where every element reveals the presence or absence of connection between two nodes. These types of encoding are generally quite simple and easy to implement. However, they also have a large amount of inconveniences as scalability, the incapability of encoding repeated structures, or permutation. Apart from direct encoding, there are some indirect encoding methods. In these methods, only some characteristics of the architecture are encoded in the chromosome. These methods have several types of representation. Firstly, the parametric representations represent the network as a group of parameters such as number of hidden layers, number of nodes for each layer, number of connections between two layers, etc. Although the parametric representation can reduce the length of the chromosome, the evolutionary algorithm performs the search within a restricted area in the search space containing all the possible architectures. Another non direct representation type is based on grammatical rules. In this system, the network is represented by a group of rules, with the shape of production rules which develop a matrix that represents the network, which has several restrictions. The growing methods represent another type of encoding. In this case, the genotype does not encode a network directly, but it contains a set of instructions for building up the phenotype. The genotype decoding consists on the execution of those instructions. With regards to the evolution of the learning rule, there are several approaches, although most of them are only based on how learning can modify or guide the evolution and also on the relationship among the architecture and the connection weights.

Graph-Based Genetic Programming

As was described, the codification type of the GP algorithm has the shape of trees. This allows the solving of a great size of different problems. It also allows the finding of solutions that other codifications, e.g. bit or real strings as used on GAs, can not find. Graph codification also allows the solving of problems that could not be solved with tree-shape codification. Few after the appearance of GP, some researchers have studied the possibility of using graphs inside GP to represent and solve these problems. As first approximations of graph-based GP, some solutions appeared that used trees with special operators with the objective of solving very specific problems, e.g. to develop stack automata or electrical circuits. In this field there are some works in which classic GP is used to develop different types of electrical circuits and analogical filters. To do this, some operators had to be created to allow GP represent so complex structures. Although the results have been very satisfactory, this kind of codification, using these operators, is very limited, and has only allowed the solving of problems in this field. Other approximations to the codification of graphs by using trees are Gruau's and Luke's, mainly used to develop ANNs with GP. These works use the operators on the GP tree to build graphs as this tree and its operators are being executed. These operators can be used, e.g, to create new nodes or to create links between nodes. These codifications are called cellular encoding or edge encoding, and have some drawbacks. First, the represented

phenotype (i.e., the obtained graph) is too dependent of the execution order of the operators of the tree. A subtree inside an individual can turn into a completely different subtree after being crossed with another individual. Therefore, it is desirable to use a crossover algorithm that preserves better the phenotype on this operation. Also, this type of codification produces a high number of interconnected nodes, which can not be very desirable on many domains. Teller describes a system called PADO which uses a stack and lineal discriminators in order to obtain parallel programs used for signal and image classification. All of these methods use special types of parallelisms in the use of graphs, and can not be considered as a natural generalization of GP to the use of graphs. In a similar way, Kanstchik uses graphs as codification system, having for this purpose an indexed memory, which is used to transfer data between nodes. These nodes are divided in two parts: action and ramification. The action part can be a constant or a function that will be executed when the node is reached during the program execution. To do this execution, this function takes its parameters from the stack, and writes its result on the stack too. After this, the ramification part chooses, by examining the top of the stack, which node will go on the execution between the nodes to which is connected the current one. Another GP system which uses graphs is called Parallel Distributed Genetic Programming (PDGP). In PDGP, programs are represented as graphs with nodes that represent program primitives and links that represent the execution flow and the results. Therefore, PDGP can be used to evolve parallel programs or to produce sequential programs with shared (or reutilized) subtrees. This application defines new crossover and mutation operators for their use with these graphs. In a new approximation to the use of graphs, this time called linear-graph GP, some special nodes were used. These nodes execute a set of sequential operations and end in a conditional ramification. As a result of this ramification, these nodes point to other nodes of the same type, allowing the pointing to a node referenced more than once. Although this study allows the working directly with graphs, this application was only created to develop graph-shaped sequential programs, and can not be used to solve more generic problems. One of the most representative works in this field is called Neural Programming. This application describes a system that uses graphs as codification type. Also, this application uses a credit system on the connections between nodes to choose better which ones will be used in the individual combinations. However, this system only works with mathematical graphs (i.e., graphs with nodes representing mathematical operations such as arithmetical, trigonometrical, etc.) because it was developed for image and signal processing. Anyway, this system is one of the most complete existing ones.

8.9.2 Model

This application uses a graph-based codification to represent ANNs in the genotype. These graphs will not contain any cycles. Due to this type of codification the

genetic operators had to be changed in order to be able to use the GP algorithm. The operators were changed in this way:

- The creation algorithm must allow the creation of graphs. This means that, at the moment of the creation of a node's child, this algorithm must allow not only the creation of this node, but also a link to an existing one in the same graph, without making cycles inside the graph.
- The crossover algorithm must allow the crossing of graphs. This algorithm works very similar to the existing one for trees, i.e. a node is chosen on each individual to change the whole subgraph it represents to the other individual. Special care has to be taken with graphs, because before the crossover there may be links from outside this subgraph to any nodes on it. In this case, after the crossover these links are updated and changed to point to random nodes in the new subgraph.
- The mutation algorithm has been changed too, and also works very similar to the GP tree-based mutation algorithm. A node is chosen from the individual and its subgraph is deleted and replaced with a new one. Before the mutation occurs, there may be nodes in the individual pointing to other nodes in the subgraph. These links are updated and made to point to random nodes in the new subgraph. These algorithms must also follow two restrictions in GP: typing and maximum height. The GP typing property means that each node will have a type and will also provide which type will have each of its children. This property provides the ability of developing structures that follow a specific grammar. The maximum height is a restriction of the complexity of the graph, not allowing the creation of very large graphs that could lead to obtaining too big ANNs with over-fitting problems. These two restrictions are applied on the genetic operators making the resulting graphs follow these restrictions. The nodes used to build ANNs with this system are the following:
  - ***A***NN. Node that defines the network. It appears only at the root of the tree. It has the same number of descendants as the network expected outputs, each of them a neuron.
  - ***n***-Neuron. Node that identifies a neuron with n inputs. This node will have 2^*n descendants. The first n descendants will be other neurons, either input or hidden ones. The second n descendants will be arithmetical sub-trees. These sub-trees represent real values. These values correspond to values of the respective connection weights of the input neurons – the first descendants – of this neuron.
  - ***n***-Input neuron. Nodes that define an input neuron which receives its activation value from the input variable n. These nodes will not have any children.
- Finally, the arithmetic operator set $\{+, -, ^*, \%\}$, where $\%$ designs the operation of protected division (returns 1 as a result if the divisor is 0). They will generate the values of connection weights (sub-trees of the n-Neuron nodes). These nodes perform operations among constants in order to obtain new values. As real values are also needed for such operations, they have to be introduced by means of the addition of random constants to the terminal set in the range $[-4, 4]$. The execution of the graph will make the creation of the ANN: each n-Neuron node will

make the creation on one neuron and links to the neurons which are connected to that, each n-Input node will connect an input neuron to another neuron, and an arithmetical subgraph will set the value of a weight. An example of this can be seen on Figure 8.36. Note that, during the neuron creation, a given neuron - either an input one or a referenced one - can be repeated several times as predecessor. In such case, there is no new input connection from that processing element, but the weight of the already existing connection will be added with the value of the new connection. Once the tree has been evaluated, the genotype turns into phenotype. In other words, it is converted into an ANN with its weights already set (thus it does not need to be trained) and therefore it can be evaluated. The evolutionary process demands the assignation of a fitness value to every genotype. Such value is the result of the evaluation of the network with the pattern set representing the problem. This result is the mean square error (MSE) of this evaluation. Nevertheless, this error value considered as fitness value has been modified in order to induce the system to generate simple networks. The modification has been made by adding a penalization value multiplied by the number of neurons of the network. In such way, and given that the evolutionary system has been designed in order to minimize the error value, when adding a fitness value, a larger network would have a worse fitness value. Therefore, the existence of simple networks is preferred as the penalization value that is added is proportional to the number of neurons of the ANN. The calculus of the final fitness will be as follows:

$$fitness = MSE + N^{*}P'$$

(1) where MSE is the mean square error of the ANN with the training pattern set, N is the number of neurons of the network and P is the penalization value for such number.

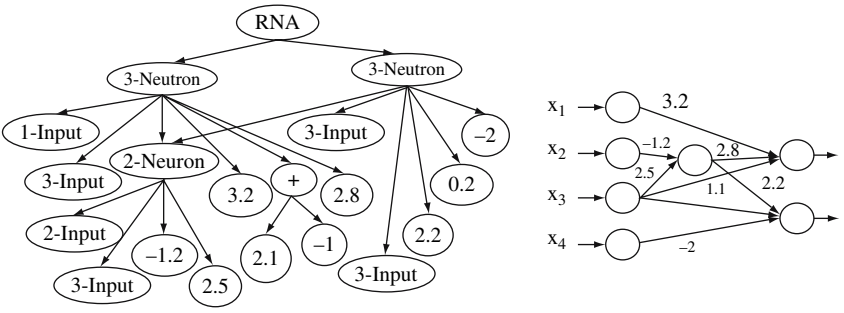


Fig. 8.36 GP Graph and its resulting network

Table 8.19 Summary of the Problem to be solved

Problem	Number of Inputs	Number of data points	Number of Outputs
Breast Cancer	9	699	1
Iris Flower	4	150	3
Heart Cleveland	13	303	1
Ionosphere	34	351	1

8.9.3 Problems to be Solved

This technique has been used for solving problems of different complexity taken from the UCI database. All these problems are knowledge-extraction problems from databases, where, taking certain features as a basis, it is intended to perform a prediction about another attribute of the database. The value that is intended to be predicted might be a diagnosis value (when using medical databases), a classification value or a prediction one. A small summary of the problems to be solved can be seen on Table 8.19.

All these databases values have been normalized between 0 and 1 and the pattern sets divided into two parts for each problem, taking the 70% of the database for training and using the remaining 30% for performing tests.

8.9.4 Results and Comparison with Other Methods

Several experiments have been performed in order to valuate the system performance. The values taken for the parameters at these experiments were the following:

- Crossover rate: 95%.
- Mutation probability: 4%.
- Selection algorithm: 2-individual tournament.
- Graph maximum height: 5.
- Maximum inputs for each neuron: 9.
- Population size: 1000 individuals.
- Penalization value: 0.00001.

To achieve these values, several experiments had to be done in order to obtain values for these parameters that would return good results to all of the problems. These problems are very different in complexity, so it is expected that these parameters give good results to many different problems. This last parameter, the penalization to the number of neurons, is important. The values range from very high (0.1) to very small (0.00001 or 0). High values only enables the creation of very small networks with a subsequent high error, and low values lead to overfitting problem.

In order to evaluate its performance, the system presented here has been compared with other ANN generation and training methods. The method 5×2cv is used by Cantú-Paz and Kamath's for the comparison of different ANN generation and training techniques based on EC tools. This application presents as results the average precisions obtained in the 10 test results generated by this method. Such values are the basis for the comparison of the technique described here with other well known ones. These algorithms used on the comparison are widely explained with detail in Cantú-Paz and Kamath's application. Such application shows the average times needed to achieve the results. Not having the same processor that was used, the computational effort needed for achieving the results can be estimated. This effort represents the number of times that the pattern file was evaluated. The computational effort for every technique can be measured using the population size, the number of generations, the number of times that the BP algorithm was applied, etc. This calculation varies for every algorithm used. All the techniques that are compared with the work are related to the use of evolutionary algorithms for ANN design. Five iterations of a 5-fold crossed validation test were performed in all these techniques in order to evaluate the precision of the networks. These techniques are connectivity matrix, pruning, parameter search and graph rewriting grammar. Table 8.20 shows a summary of the number of neurons used in Cantú-Paz and Kamath's application in order to solve the problems that were used with connectivity matrix and pruning techniques. The epoch number of the BP algorithm, when used, is also indicated here.

Table 8.21 shows the parameter configuration used by these techniques. The execution was stopped after 5 generations with no improvement or after total generations.

In this application, the procedures for design and training are performed simultaneously, and therefore, the times needed for designing as well as for evaluating

Table 8.20 Architectures Used

	Inputs	Hidden	Outputs	BP Epochs
Breast Cancer	9	5	1	20
Iris Flower	4	5	3	80
Heart Cleveland	26	5	1	40
Ionosphere	34	10	1	40

Table 8.21 Parameters of the techniques used during comparison

	Matrix	Prunning	Parameters	Grammar
Chromosome Length (L)	N	N	36	256
Population Size	$3\sqrt{}$L	$3\sqrt{}$L	25	64
Crossover points	L/10	L/10	2	L/10
Mutation rate	1/L	1/L	0.04	0.004

$N = (hidden + output)^* input + output^* hidden$

the network are combined. Most of the techniques used for the ANN development are quite costly, due in some cases to the combination of training with architecture evolution. The technique described here is able to achieve good results with a low computational cost and besides, the added advantage is that, not only the architecture and the connectivity of the network are evolved, but also the network itself undergoes an optimization process.

8.9.5 Concluding Remarks

This application presents a technique for ANN generation with GP based on graphs. To develop this system, the evolutionary operators had to be modified in order to be able to work with graphs instead of trees. This system has been compared to a set of different techniques that use evolutionary algorithms for ANN design and training. The conclusion of such comparison is that the results of 5×2cv tests using this method are not only comparable to those obtained with other methods, but also better than them in most of the cases. It should be borne in mind that if the parameters of the system had been adapted to every problem to be solved the results would have been better. However, the parameters used have been the same for all the problems because it is intended to find a parameter set useful for any problem, and therefore there is no need of human participation. In such way, it can be stated that even without human participation, this method can improve the results of other algorithms. This system has another advantage over other methods of ANN generation since -after a short analysis by the system- it is possible to differentiate the variables that are not relevant for problem solving, as they would be not present at the ANN. This is an important feature, since it gives new knowledge about the problem being solved.

Chapter 9
Applications of Parallel Genetic Algorithm

Learning Objectives: On completion of this chapter the reader will have knowledge on PGA applications such as:

- Timetabling Problem
- Assembling DNA Fragments
- Job Shop Scheduling Problems
- Graph coloring Problem
- Ordering Problem

9.1 Timetabling Problem

9.1.1 Introduction

The problem of creating a valid timetable involves scheduling classes, teachers and rooms into a fixed number of periods, in such a way that no t*eacher, class or room* is used more than once per period. For example, if a class must meet twice a week, then it must be placed in two different periods to avoid a clash. A class consists of a number of students. It is assumed that the allocation of students into classes is fixed, and that classes are disjoint, that is, they have *no students in common*. In this scheme, a correct timetable is one in which a class can be scheduled concurrently with any other class. In each period a class is taught a *subject*. It is possible for a subject to appear more than once in a period. A particular combination of a teacher, a subject, a room and a class is called a tuple. A tuple may be required more than once per week. Thus, the timetabling problem can be phrased as scheduling a number of tuples such that a teacher, class or room does not appear more than once per period.

The task of combining a class, teacher and room combination into a tuple is handled as a separate operation, and is not the subject of this application. Tuples are formed with knowledge of the requirements of the various classes and teachers, as

well as information on room availability. It is convenient to partition the problem in this way as it reduces the complexity of the scheduling task. In many cases it is possible to form the tuples without the need to use an optimization scheme because the relationship between classes, teachers and rooms is often fixed.

It is possible to define an *objective* or *cost function* for evaluating a given timetable. This function is an arbitrary measure of the quality of the solution. A convenient cost function calculates the number of clashes in any given timetable. An acceptable timetable has a cost of 0. The optimization problem becomes one of minimizing the value of this cost function. The cost of any period can be expressed as the sum of three components corresponding to a class cost, a teacher cost and a room cost. It is not strictly necessary to sum the components, providing they can be combined to reflect the quality of the solution. However, by using a sum, it is easy to weight the various costs so that one may be made more important than the others. In this way the optimization process can be guided towards a solution in which some types of clash are more important than others.

The class cost is the number of times each of the classes in the period appears in that period, less one if it is greater than zero. Thus, if a class does not appear or appears once in a period then the cost of that class is zero. If it appears many times the class cost for that class is the number of times less one. The class cost for a period is the sum of all class costs. The same computation applies for teachers and rooms. The cost of the total timetable is the sum of the period costs. Therefore, any optimization technique should aim to find a configuration with the lowest possible cost.

9.1.2 Applying Genetic Algorithms to Timetabling

A timetable can be represented by a fixed set of tuples, each of which contains a class number, a teacher number and a room number. The scheduling algorithm must assign a period number to each of these tuples such that a given class, teacher or room does not appear more than once in a period.

In order to use a genetic algorithm to solve this problem, it is necessary to devise a representation for a timetable that maps onto chromosomes. Each chromosome is composed of genes, each of which represents some property of the individual timetable. Figure 9.1 shows a sample timetable as a collection of tuples. Each period

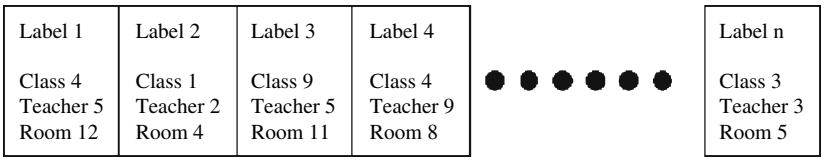


Fig. 9.1 Representing the timetable problem using tuples

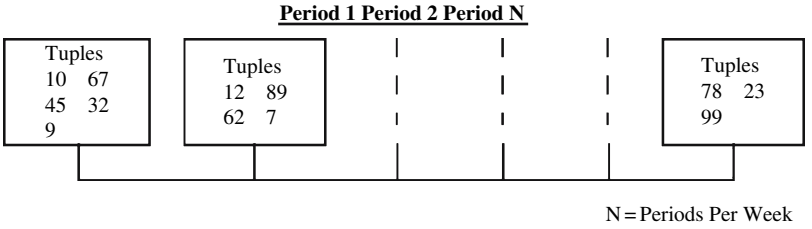


Fig. 9.2 Mapping tuples onto periods

contains a number of tuples, identified by their *label*. The values of the fields of the tuple would contain valid class, teacher and room numbers.

In Figure 9.2 the data has been mapped onto periods, and shows many tuples packed into each period. The cost of a timetable can then be computed by adding up the number of clashes in each period. The good and bad attributes of a timetable are related to the quality of the packing of tuples into periods, rather than the absolute period location of any one tuple. Thus, the genetic representation must preserve good packings, remove bad ones and allow new packings to form as the population evolves.

Figure 9.3 shows one possible mapping of tuples onto chromosomes. In this scheme each period *represents* a chromosome. Using this mapping allows good packings to be preserved between generations, because corresponding periods are mated, as described shortly. A number of representations were discarded because they did not allow packings to propagate between generations.

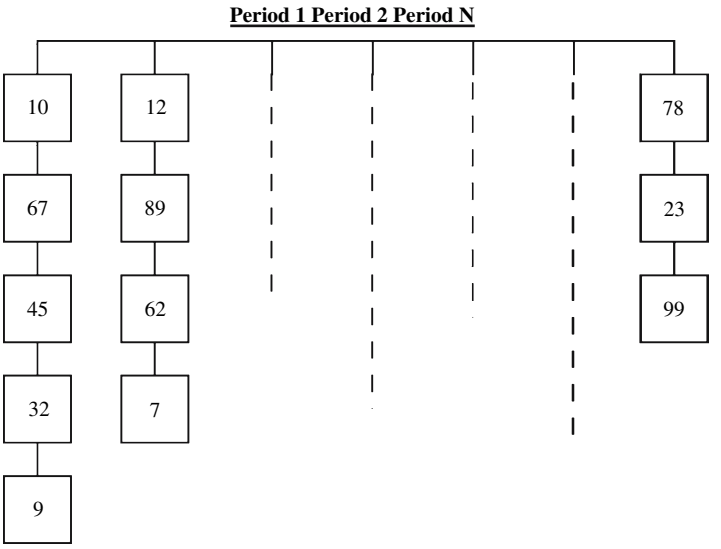


Fig. 9.3 Representing tuples using chromosomes

During mating a child is created which has some of the genes from one parent, and the remainder from the other. This process is called cross-over, and is implemented as a random selection of genes from each parent. During mating, the child may be mutated. It is mutation, which introduces diversity into the population, and allows new class, teacher and room mappings to evolve. Mutation is performed on a randomly selected gene, which is then mutated in a random way. In the timetable problem, this corresponds to altering the period in which a tuple is located.

Figure 9.4 shows the process of mating two timetables to produce a new timetable. The diagram only shows the cross over of one period. Each period is crossed over independently. In this example, a random crossover site is chosen, and the new period in the child is constructed with the first 3 genes of parent 1, and the last 2 of parent 2. The same process creates each period of the child. If the cross-over site is at either end of the chromosome, then all of that period is taken from only one parent. The fitness of the new child can be computed incrementally as the change in cost of the period.

Figure 9.5 shows how the child can receive further mutation of its mapping. In this example, a randomly chosen tuple from a randomly chosen period is moved to another randomly chosen period.

Genetic algorithms require a measure of the fitness of an individual in order to determine which individuals in the population survive. The higher the fitness, the more likely the individual will survive. The cost measure developed for the timetabling problem represents a high quality solution with a minimal cost value (a value of 0

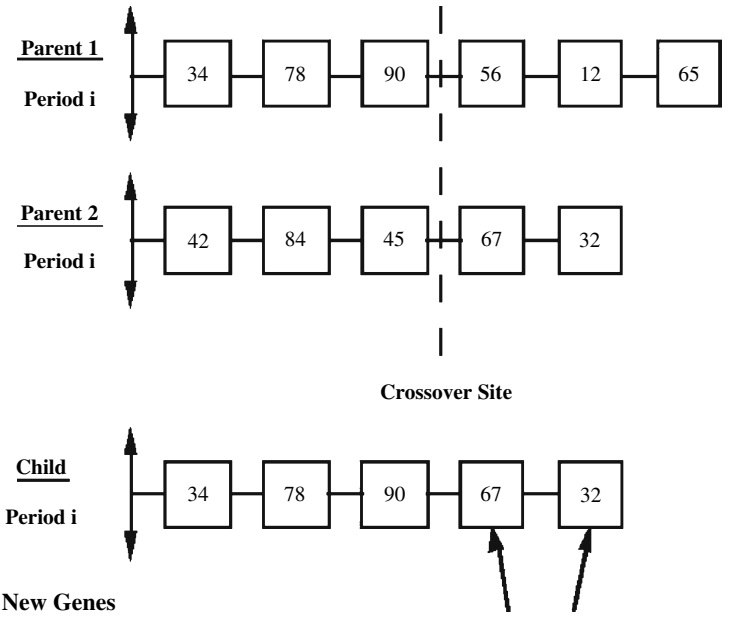


Fig. 9.4 Mating two timetables

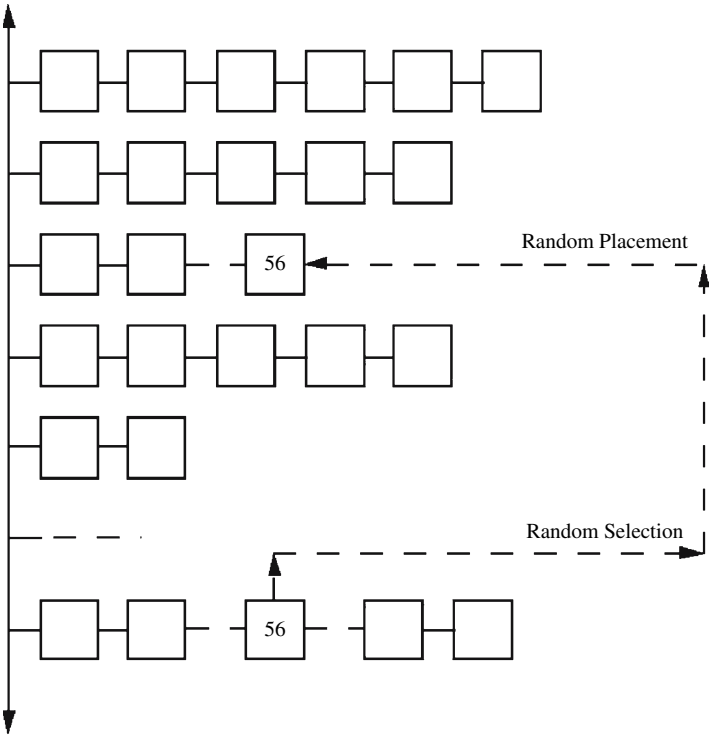


Fig. 9.5 Mutating a child

is optimal). Thus, a fitness value is computed from the cost of a timetable by subtracting the cost from a fitness ceiling. In this way, an increased cost generates a lower fitness value. Fitness values are further scaled so that unfit individuals have a negative fitness value, and are then removed from the population.

The representation scheme was chosen because it allows the good properties of a timetable to propagate from one generation to the next. However, a problem with the reproduction scheme described above is that after a mating, the contents of the timetable have been altered, and no longer reflect the original data set. This problem is known as the label replacement problem, and is caused because the cross over takes tuples period by period, rather than tuple by tuple. Thus, the disadvantage is that the child no longer contains the correct set of tuples. This problem is illustrated in Figure 9.6, where label 26 has been duplicated in the child.

The solution to the problem of loss of tuples is to use a label replacement algorithm. After a child has been created, it is modified to restore the original tuple set. Thus, a tuple, which has been lost through mating, is restored in a random location, effectively mutating the child chromosome. Similarly, copies of duplicated tuples are removed. Label replacement can be seen as a form of mutation because it introduces extra diversity into the population.

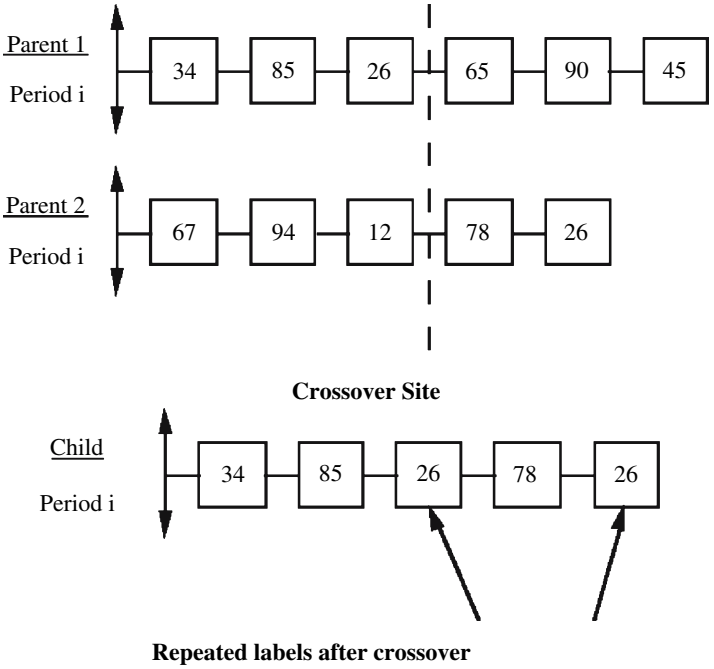


Fig. 9.6 Problem caused by loss of labels

It is worth considering why label replacement does not occur in biological mating. The reason is that when cross-over occurs, the child inherits one set of genes from one parent, and the exact complement of genes from the other parent. The scheme works because a gene represents some quality of the individual, such as blue eyes. However, the representation chosen for the timetables means that the genes represent some packing of tuples in the individual, but not the exact identity of the tuple. Thus, after crossover, the child has some genes duplicated, whilst losing others.

9.1.3 A Parallel Algorithm

Biological mating is highly concurrent. In practice, individuals mate with little regard to the rest of the population. However, many computer simulations of evolution are sequential: the simulation proceeds by sequentially creating a new population as a sequence of mating. The algorithm discussed in this section has been designed for a shared memory multiprocessor. The most obvious target for concurrency is in the creation of the new population, because each mating operation is largely independent of any other. Another source of concurrency is the actual mating operation, which requires independent manipulation on an individual. However, the

population size is typically much larger than the number of processors available in shared memory machines, and thus it is necessary only to consider the first source of concurrency.

The genetic algorithm can be summarized as follows:

```
while number of generations < limit & no perfect
individual do
for each child in the new population do
    choose two living parents at random from old
    population
    create an empty child
    for each period of the parents do
        mate corresponding periods
        copy new child period to corresponding position
        in child od
    repair lost & duplicated labels
    apply mutation to randomly selected period & tuple
    measure fitness of individual.
    if fitness < minimum allowed fitness (based on
    fitness scaling) then
        set child status to born dead
    else
        set child status to living fi
od
old population=new population
od
```

The algorithm is viewed from the perspective of the child because this allows random selection of parents. The mating is performed with a randomly chosen crossover site within the period, and is done for each period of the individual. Mutation is performed on randomly selected periods and tuples, and occurs with some specified probability.

Figure 9.7 shows a template for a parallel mating process. Creation of each child is by random selection of parents, and then random mutation. Instead of one sequential piece of code creating all the children in the new population, a number of workers are spawned and each is then responsible for a fixed number of the children. It is important to minimize interprocess synchronization to maximize the speedup. Since the parents are used in a read-only mode no synchronization is required when the parents are accessed. Further, no synchronization is required on creation of children because only one worker creates each child, and the new population is only written once within a generation. In a shared memory machine it is possible for the workers to independently write into pre-allocated slots of the new population. Barrier level synchronization is required at the end of each generation.

In order to provide a deterministic execution history, each worker uses a separately seeded pseudo-random number generator. Thus, two executions of the program with the same numbers of workers and the same initial seeds generate the same result.

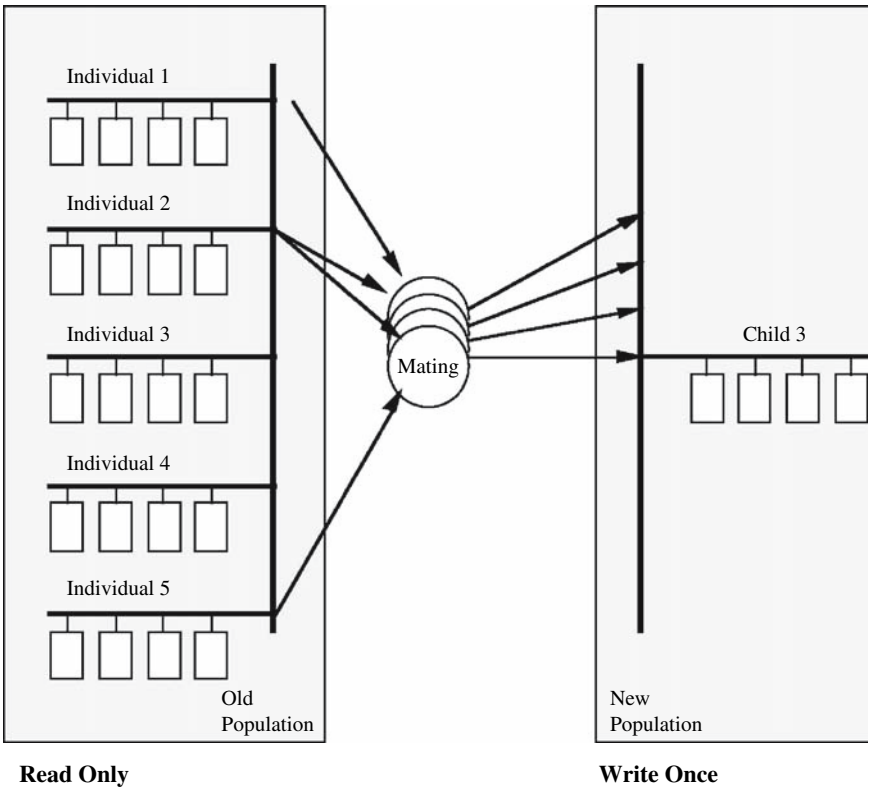


Fig. 9.7 A Parallel Mating Scheme

9.1.4 Results

In this section results of 9 data sets are presented. Table 9.1 shows the number of combinations of classes, teachers and rooms that were scheduled, as well as the number of classes, teachers and rooms in the data set. In all cases, the week is

Table 9.1 Performance of sequential genetic algorithm

Data	#Tuples	#Classes	#Teachers	#Rooms	Final Cost	# Gen
1	100	15	15	15	0	7
2	150	15	15	15	0	31
3	200	15	15	15	0	102
4	250	15	15	15	0	198
5	300	15	15	15	0	445
6	90	3	3	3	0	307
7	120	4	4	4	0	4098
8	150	5	5	5	0	3723
9	180	6	6	6	0	42973

Table 9.2 Speedup of tests on data set 7

Data	#Tuples	Population	Time (secs)/Processors					Peak Speedup
			1	2	5	10	15	
7	120	120	752	376	197	108	81	9.3 @ 15 procs
7	120	200	1886	921	431	205	–	9.2 @ 10 procs

composed of 30 periods. The final cost is the number of clashes that remained in the best individual of the final population. The number of generations required to solve the problems is shown.

The results in Table 9.1 show that the genetic algorithm is capable of finding the global minima in all cases. Data sets 6 through 9, whilst small, are highly constrained, and thus are quite difficult to solve. The population size was 100.

In Table 9.2 the speedup attained from a parallel implementation of the program is presented. The code was written in Pascal and run on an Encore Multimax shared memory multiprocessor. The times were taken for a fixed number of generations (100) so that the effects of relative solution quality could be ignored. The table shows the execution time in seconds when 1, 2, 5, 10 and 15 processors were assigned to the mating algorithm. The speedup is defined as the sequential time divided by the best parallel time, thus ideal speedup should be equal to the number of processors used to achieve the best parallel time. Two different population sizes were chosen. The results show that the speedup is less than optimal. This is because there are a few sections of code in the critical path of the program, which have not yet been parallelized.

9.1.5 Conclusion

It is shown that genetic algorithms can be applied to this complex scheduling problem. The results shown in the previous section indicate that the technique is quite powerful in finding the global minimum from an enormous search space.

Because the genetic algorithms are quite slow to execute, a solution was developed for execution on a parallel processor. Because the process of breeding is inherently parallel, quite good speedups are observed over sequential execution of the algorithm.

9.2 Assembling DNA Fragments with a Distributed Genetic Algorithm

9.2.1 Introduction

DNA fragment assembly is a technique that attempts to reconstruct the original DNA sequence from a large number of fragments, each one having several hundred

base-pairs (bps) long. The DNA fragment assembly is needed because current technology, such as gel electrophoresis, cannot directly and accurately sequence DNA molecules longer than 1000 bases. However, most genomes are much longer. For example, a human DNA is about 3.2 billion nucleotides in length and cannot be read at once.

The following technique was developed to deal with this limitation. First, the DNA molecule is amplified, that is, many copies of the molecule are created. The molecules are then cut at random sites to obtain fragments that are short enough to be sequenced directly. The overlapping fragments are then assembled back into the original DNA molecule. This strategy is called *shotgun sequencing*. Originally, the assembly of short fragments was done by hand, which is inefficient and error-prone. Hence, a lot of effort has been put into finding techniques to automate the shotgun sequence assembly. Over the past decade a number of fragment assembly packages have been developed and used to sequence different organisms. The most popular packages are PHRAP, TIGR assembler, STROLL, CAP3, Celera assembler, and EULER. These packages deal with the previously described challenges to different extent, but none of them solves them all. Each package automates fragment assembly using a variety of algorithms. The most popular techniques are greedy-based. This application reports on the design and implementation of a parallel-distributed genetic algorithm to tackle the DNA fragment assembly problem.

9.2.2 The DNA Fragment Assembly Problem

Here is a vivid analogy to the fragment assembly problem: "Imagine several copies of a book cut by scissors into thousands of pieces, say 10 millions. Each copy is cut in an individual way such that a piece from one copy may overlap a piece from another copy. Assume one million pieces lost and remaining nine million are splashed with ink, try to recover the original text." The DNA target sequence is thought of as being the original text and the DNA fragments are the pieces cut out from the book. To further understand the problem, the following basic terminology has to be known:

Fragment: A short sequence of DNA with length up to 1000 bps.
Shotgun data: A set of fragments.
Prefix: A substring comprising the first n characters of fragment f.
Suffix: A substring comprising the last n characters of fragment f.
Overlap: Common sequence between the suffix of one fragment and the prefix of another fragment.
Layout: An alignment of collection of fragments based on the overlap order.
Contig: A layout consisting of contiguous overlapping fragments.
Consensus: A sequence derived from the layout by taking the majority vote for each column of the layout.

The distribution of the coverage is considered to measure the quality of a consensus. Coverage at a base position is defined as the number of fragments at that

position. It is a measure of the redundancy of the fragment data. It denotes the number of fragments, on average, in which a given nucleotide in the target DNA is expected to appear. It is computed as the number of bases read from fragments over the length of the target DNA.

$$Coverage = \frac{\sum_{i=1}^{n} length\ of\ the\ fragment\ i}{target\ sequence\ length} \tag{9.1}$$

where n is the number of fragments. TIGR uses the coverage metric to ensure the correctness of the assembly result. The coverage usually ranges from 6 to 10 [10]. The higher the coverage, the fewer the gaps are expected, and the better the result.

9.2.3 DNA Sequencing Process

To determine the function of specific genes, scientists have learned to read the sequence of nucleotides comprising a DNA sequence in a process called DNA sequencing. The fragment assembly starts with breaking the given DNA sequence into small fragments. To do that, multiple exact copies of the original DNA sequence are made. Each copy is then cut into short fragments at random positions. These are the first three steps depicted in Figure 1 and they take place in the laboratory. After the fragment set is obtained, traditional assemble approach is followed in this order: overlap, layout, and then consensus. To ensure that enough fragments overlap, the reading of fragments continues until the coverage is satisfied. These steps are the last three steps in Figure 9.8. In the following section, a brief description of each of the three phases, namely overlap, layout, and consensus is given.

Overlap Phase - Finding the overlapping fragments. This phase consists in finding the best or longest match between the suffix of one sequence and the prefix of another. In this step, all possible pairs of fragments are compared to determine their similarity. Usually, the dynamic programming algorithm applied to semi global alignment is used in this step. The intuition behind finding the pair wise overlap is that fragments with a significant overlap score are very likely next to each other in the target sequence.

Layout Phase - Finding the order of fragments based on the computed similarity score. This is the most difficult step because it is hard to tell the true overlap due to the following challenges:

- Unknown orientation: After the original sequence is cut into many fragments, the orientation is lost. The sequence can be read in either $5'$ to $3'$ or $3'$ to $5'$. One does not know which strand should be selected. If one fragment does not have any overlap with another, it is still possible that its reverse complement might have such an overlap.
- Base call errors: There are three types of base call errors: substitution, insertion, and deletion errors. They occur due to experimental errors in the electrophoresis

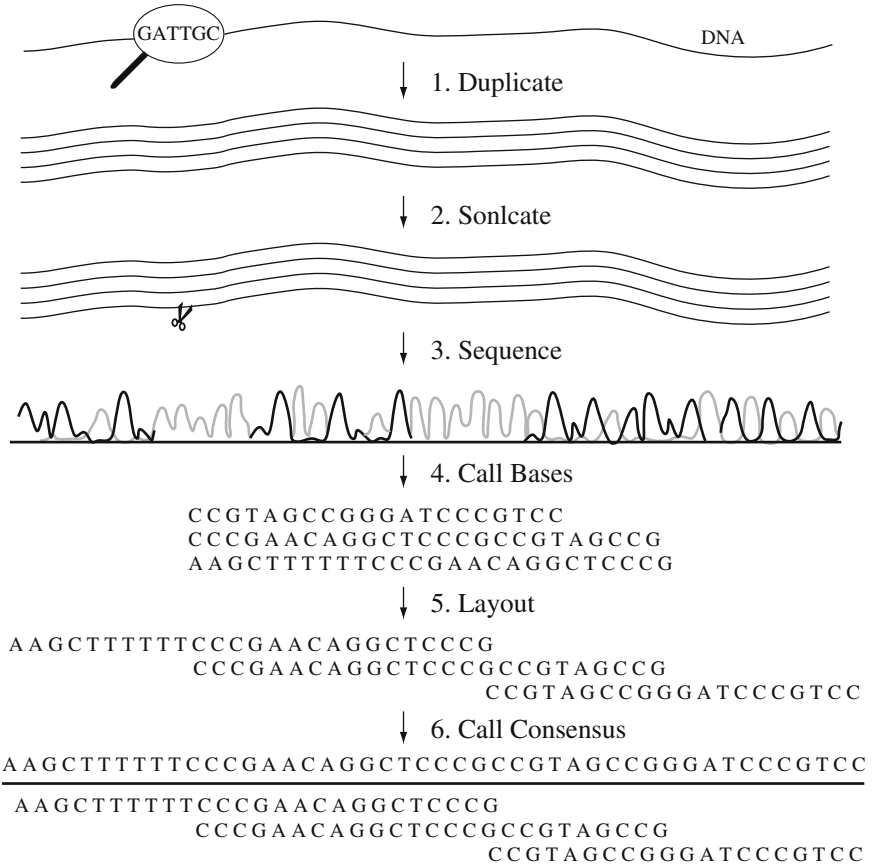


Fig. 9.8 Graphical representation of DNA sequencing and assembly

procedure. Errors affect the detection of fragment overlaps. Hence, the consensus determination requires multiple alignments in high coverage regions.

- Incomplete coverage: It happens when the algorithm is not able to assemble a given set of fragments into a single contig.
- Repeated regions: Repeats are sequences that appear two or more times in the target DNA. Repeated regions have caused problems in many genome-sequencing projects, and none of the current assembly programs can handle them perfectly.
- Chimeras and contamination: Chimeras arise when two fragments that are not adjacent or overlapping on the target molecule join together into one fragment. Contamination occurs due to the incomplete purification of the fragment from the vector DNA.

After the order is determined, the progressive alignment algorithm is applied to combine all the pairwise alignments obtained in the overlap phase.

Consensus Phase - Deriving the DNA sequence from the layout. The most common technique used in this phase is to apply the majority rule in building the consensus.

Example: An example of the fragment assembly process is given below:

Given a set of fragments F1 = GTCAG, F2 = TCGGA, F3 = ATGTC, F4 = CGGATG, assume the four fragments are read from $5'$ to $3'$ direction. First, it is necessary to determine the overlap of each pair of the fragments by the using semiglobal alignment algorithm. Next, the order of the fragments is determined based on the overlap scores, which are calculated in the overlap phase. Suppose if the order is F2 F4 F3 F1. Then, the layout and the consensus for this example can be constructed as follows:

F2 -> TCGGA F4 -> CGGATG F3 -> ATGTC F1 -> GTCAG
Consensus-> TCGGATGTCAG

In this example, the resulting order allows to build a sequence having just one contig. Since finding the exact order takes a huge amount of time, a heuristic such as Genetic Algorithm can be applied in this step. In the following section, it is illustrated as to how the Genetic Algorithm is implemented for the DNA fragment assembly problem.

9.2.4 DNA Fragment Assembly Using the Sequential GA

GA uses the concept of survival of the fittest and natural selection to evolve a population of individuals over many generations by using different operators: selection, crossover, and mutation. As the generations are passed along, the average fitness of the population is likely to improve. Genetic Algorithm can be used for optimization problems with multiple parameters and multiple objectives. It is commonly used to tackle NP-hard problems such as the DNA fragment assembly and the Travelling Salesman Problem (TSP). NP-hard problems require tremendous computational resources to solve exactly. Genetic Algorithms help to find good solutions in a reasonable amount of time. Next, the sequential GA for the fragment assembly problem is presented.

1 Randomly generate the initial population of fragment orderings.
2 Evaluate the population by computing fitness.
3 while (NOT termination condition)

(a) Select fragment orderings for the next generation through ranking selection
(b) Alter population by
    i. applying the crossover operator
    ii. applying the mutation operator
    iii. re-evaluate the population.

9.2.5 Implementation Details

Details about the most important issues of the implementation are given below.

Population Representation

The permutation representation with integer number encoding is used. A permutation of integers represents a sequence of fragment numbers, where successive fragments overlap. The population in this representation requires a list of fragments assigned with a unique integer ID. For example, 8 fragments will need eight identifiers: 0, 1, 2, 3, 4, 5, 6, 7. The permutation representation requires special operators to make sure that always legal (feasible) solutions are obtained. In order to maintain a legal solution, the two conditions that must be satisfied are (1) all fragments must be presented in the ordering, and (2) no duplicate fragments are allowed in the ordering. For example, one possible ordering for 4 fragments is 3 0 2 1. It means that fragment 3 is at the first position and fragment 0 is at the second position, and so on.

Population Size

A fixed size population is used to initialize random permutations.

Program Termination

The program can be terminated in one of two ways. Either the maximum number of generations can be specified or the algorithm can be stopped when the solution is no longer improving.

Fitness Function

A fitness function is used to evaluate how good a particular solution is. It is applied to each individual in the population and it should guide the genetic algorithm towards the optimal solution. In the DNA fragment assembly problem, the fitness function measures the multiple sequences alignment quality and finds the best scoring alignment. Parsons, Forrest, and Burks mentioned two different fitness functions.

Fitness function F1 sums the overlap score for adjacent fragments in a given solution. When this fitness function is used, the objective is to maximize such a score. It means that the best individual will have the highest score.

$$F1(l) = \sum_{i=0}^{n-2} w \cdot (f[i]f[i+1]) \tag{9.2}$$

Fitness function F2-not only sums the overlap score for adjacent fragments, but it also sums the overlap score for all other possible pairs.

$$F2(l) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} |i-j| \times w \cdot (f[i]f[j]) \tag{9.3}$$

This fitness function penalizes solutions in which strong overlaps occur between non-adjacent fragments in the layouts. When this fitness function is used, the objective is to minimize the overlap score. It means that the best individual will have the lowest score.

The overlap score in both F1 and F2 is computed using the semiglobal alignment algorithm.

Recombination Operator

Two or more parents are recombined to produce one or more offspring. The purpose of this operator is to allow partial solutions to evolve in different individuals and then combine them to produce a better solution. It is implemented by running through the population and for each individual, deciding whether it should be selected for crossover using a parameter called *crossover rate* (*Pc*). A crossover rate of 1.0 indicates that all the selected individuals are used in the crossover. Thus, there are no survivors. However, empirical studies have shown that better results are achieved by a crossover rate between 0.65 and 0.85, which implies that the probability of an individual moving unchanged to the next generation ranges from 0.15 to 0.35.

For the experimental runs, the order-based crossover (OX) and the edge-recombination crossover (ER) are used. These operators were specifically designed for tackling problems with permutation representations.

The order-based crossover operator first copies the fragment ID between two random positions in Parent1 into the offspring's corresponding positions. Then the rest of the fragments from Parent2 is copied into the offspring in the relative order presented in Parent2. If the fragment ID is already present in the offspring, then that fragment is skipped. The method preserves the feasibility of every string in the population.

Edge recombination preserves the adjacencies that are common to both parents. This operator is appropriate because a good fragment ordering consists of fragments that are related to each other by a similarity metric and should therefore be adjacent to one another. Parsons defines edge recombination operator as follows:

1 Calculate the adjacencies.
2 Select the first position from one of the parents, call it *s*.
3 Select s' in the following order until no fragments remain:

   (a) s' adjacent to s is selected if it is shared by both parents.
   (b) s' that has more remaining adjacencies is selected.
   (c) s' is randomly selected if it has an equal number of remaining adjacencies.

Mutation Operator

This operator is used for the modification of single individuals. Mutation operator is necessary for maintaining diversity in the population. Mutation is implemented by running through the whole population and for each individual, deciding whether to select it for mutation or not, based on a parameter called *mutation rate* (*Pm*). For the experimental runs, swap mutation operator is used. This operator randomly selects two positions from a permutation and then swaps the two fragment positions. Since this operator does not introduce any duplicate number in the permutation, the solution it produces is always feasible. Swap mutation operator is suitable for permutation problems like ordering fragments.

Selection Operator

The purpose of the selection is to weed out the bad solutions. It requires a population as a parameter, processes the population using the fitness function, and returns a new population. The level of the selection pressure is very important. If the pressure is too low, convergence becomes very slow. If the pressure is too high, convergence will be premature to a local optimum.

n this application, ranking selection mechanism is used. Here the GA first sorts the individual based on the fitness and then selects the individuals with the best fitness score until the specified population size is reached. Note that the population size will grow whenever a new offspring is produced by crossover or mutation. The use of ranking selection is preferred over other selections such as fitness proportional selection.

9.2.6 DNA Fragment Assembly Problem using the Parallel GA

The first part of this section describes the parallel model of GA, while the second part presents the software used to implement that model.

Parallel Genetic Algorithm

A parallel GA (PGA) is an algorithm having multiple component GAs, regardless of their population structure. A component GA is usually a traditional GA with a single population. Its algorithm is augmented with an additional phase of *communication* code so as to be able to convey its result and receive results from the other components.

Different parallel algorithms differ in the characteristics of their elementary heuristics and in the communication details. In this application, a kind of decentralized distributed search is chosen because of its popularity and because it can be

easily implemented in clusters of machines. In this parallel implementation separate subpopulations evolve independently in a ring with sparse exchanges of a given number of individuals with a certain given frequency as seen in Figure 9.9. The selection of the emigrants is through binary tournament in the genetic algorithms, and the arriving immigrants replace the worst ones in the population only if the new ones is better than this current worst individuals. MALLBA, used in this application is explained below.

The MALLBA Project

The MALLBA research project is aimed at developing a library of algorithms for optimization that can deal with parallelism in a user-friendly and, at the same time, efficient manner. Its three target environments are sequential, LAN and WAN computer platforms. All the algorithms described in the next section are implemented as *softwar*e *skeleton*s (similar to the concept of software pattern) with a common internal and public interface. This permits fast prototyping and transparent access to parallel platforms.

MALLBA skeletons distinguish between the concrete problem to be solved and the solver technique. Skeletons are generic templates to be instantiated by the user with the features of the problem. All the knowledge related to the solver method (e.g., parallel considerations) and its interactions with the problem are implemented by the skeleton and offered to the user. Skeletons are implemented by a set of *require*d and *provide*d C++classes that represent an abstraction of the entities participating in the solver method:

- Provided Classes: They implement internal aspects of the skeleton in a problem-independent way. The most important *provide*d classes are Solver (the algorithm) and SetUpParams (setup parameters).
- Required Classes: They specify information related to the problem.
- Each skeleton includes the Problem and Solution required classes that encapsulate the problem-dependent entities needed by the solver method. Depending on the skeleton other classes may be required.

Therefore, the user of a MALLBA skeleton only needs to implement the particular features related to the problem. This speeds considerably the creation of new

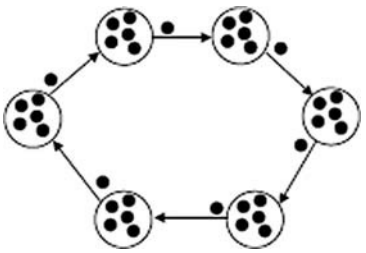


Fig. 9.9 Graphical representation of the parallel dGA

algorithms with minimum effort, especially if they are built up as combinations of existing skeletons (*hybrids*).

The infrastructure used in the MALLBA project is made of communication networks and clusters of computers located at the Spanish universities of Malaga, La Laguna and UPC in Barcelona. These nodes are interconnected by a chain of Fast Ethernet and ATM circuits. The MALLBA library is publicly available at http://neo.lcc.uma.es/mallba/easy-mallba/index.html.

By using this library, it was able to perform a quick coding of algorithmic prototypes to cope with the inherent difficulties of the DNA fragment assembly problem.

9.2.7 Experimental Results

A target sequence with accession number BX842596 (GI 38524243) was used in this application. It was obtained from the NCBI web site (http://www.ncbi.nlm.nih.gov). It is the sequence of a Neurospora crassa (common bread mold) BAC, and is 77,292 base pairs long. To test and analyze the performance of third algorithm, two problem instances are generated with GenFrag. The first problem instance, 842596 4, contains 442 fragments with average fragment length of 708 bps and coverage 4. The second problem instance, 842596 7, contains 733 fragments with average fragment length of 703 bps and coverage 7.

Each assembly result is evaluated in terms of the number of contigs assembled and the percentage similarity of assembled regions with the target sequence. Since fragments are obtained from a known target sequence, assembled consensus sequence can be compared with the target.

Sequential GA and several distributed GAs (having 2, 4, and 8 islands) are used to solve this problem. Since the results of the GA vary depending on the different parameter settings, this section first discusses how the parameters affect the results and the performance of the GA.

Analysis of the Algorithm

The effects of the fitness function, crossover operator, population size, operator rates and migration configuration for the distributed GA (dGA) are studied. In this analysis different runs of the GA are performed in the following manner: change one GA parameter of the basic configuration while keeping the other parameters to the same value. The basic setting uses F1 (Eq.9.2) as fitness function and the order-based crossover as recombination operator. The whole population is composed of 512 individuals. In dGA, each island has a population of $512/n$, where n is the number of islands. Migration occurs in a unidirectional ring manner, sending one single randomly chosen individual to the neighbor subpopulation. The target population incorporates this individual only if it is better than its presently worst solution. The migration step is performed every 20 iterations in every island in an asynchronous way. All runs were performed on a Pentium 4 at 2.8 GHz linked by a Fast

Table 9.3 Basic configuration

Independent runs	30
Popsize	512
Fitness function	F1
Crossover	OX (0.7)
Mutation	Swap (0.2)
Cutoff	30
Migration frequency	20
Migration rate	1
Instance	38524243_4.dat

Ethernet communication network. The parameter values are summarized in Table 9.3. 30 independent runs of each experiment were performed.

Function Analysis

In this problem, choosing an appropriate fitness function is an open research line, since it is difficult to capture the dynamics of the problem into a mathematical function. The results of the runs are summarized in Table 9.4. The table shows the fitness of the best solution obtained (b), the average fitness found (f), average number of evaluations (e), and average time in seconds (t). Recall that F1 is a maximization function while F2 is a minimization one. The conclusion is that, while F2 takes longer than F1, both functions need the same number of evaluations to find the best solution (differences are not statistically significant in sequential and distributed $(\mathrm{n} = 2)$ versions). This comes as no surprise, since F2 has a quadratic complexity while the complexity of F1 is linear. In fact, this amounts to an apparent advantage of F1, since it provides a lower complexity compared to F2 while needing a similar effort to arrive to similar or larger degrees of accuracy. Also, when distributed ($\mathrm{n} = 4$ or $\mathrm{n} = 8$) F1 does allow for a reduction in the effort, while F2 seems not to profit from a larger number of machines.

The number of contigs is used as the criterion to judge the quality of the results. As it can be seen in Table 9.5, F1 performs better than F2 since it produces fewer contigs.

Table 9.4 Results with F1 and F2 fitness functions

		F1				F2			
		b	f	e	t	b	f	e	t
Sequential		26358	24023	808311	56	55345800	58253990	817989	2.2e+03
LAN	n = 2	98133	86490	711168	25	58897100	61312523	818892	1.1e+03
	n = 4	75824	66854	730777	14	66187200	63696853	818602	5.4e+02
	n = 8	66021	56776	537627	6.3	77817700	79273330	817638	2.7e+02

Table 9.5 Function analysis 3: best contigs

		F1 Contig	F2 Contig
Sequential		6	8
	n = 2	6	7
LAN	n = 4	6	6
	n = 8	6	7

Crossover Operator Analysis

In this subsection the effectiveness of two recombination operators are analyzed: the order-based crossover and the edge recombination. Table 9.6 summarizes the results obtained using these operators. A close look at the columns devoted to running times reveals that ER is slower than the OX operator. This is due to the fact that ER preserves the adjacency present in the two parents, while OX does not. Despite the theoretical advantage of ER over OX, it is noticed that the GA performs equally with the order-based crossover operator as with the edge recombination, since it computes higher fitness scores for two out of two cases for the two operators (recall that F1 is a maximization function). OX operator is much faster at an equivalent accuracy.

Population Size Analysis

In this subsection the influence of the population in the algorithms are studied. Table 9.7 shows the average fitness score and the average time for several population sizes. As can be seen in the table, small population sizes lead to fast convergence to low average fitness values. The best average fitness in this set of experiments is obtained with a population of 512 individuals. For population sizes larger than 512 the execution time is increased while the average fitness does not improve. This observation shows that a population size of 512 might be optimal.

Operator Rate Analysis

Next the effect of the operator rates in the GA behavior is analyzed. Table 9.8 summarizes the results using different combinations of crossover rates (*Pc*) and

Table 9.6 Crossover Operator Analysis

		OX				ER			
		b	f	e	t	b	f	e	t
Sequential		26358	24023	808311	56	26276	23011	801435	2.01e3
	n = 2	96133	86490	711168	25	99043	84238	789685	1.1e3
LAN	n = 4	75824	66854	730777	14	73542	65893	724058	5.3e2
	n = 8	66021	56776	537627	6.3	62492	53628	557128	1.9e2

mutation rates (*Pm*). The findings show that the fitness values tend to increase as the crossover and mutation rate increase. The mutation operator is very important in this problem, since when the algorithm does not apply mutation ($Pm = 0.0$), the population converges too quickly and the algorithm yields very bad solutions.

Migration Policy Analysis

This analysis is completed by examining the effects of the migration policy. More precisely, the influence of the migration rate (*rate*) and the migration frequency (*freq*) is studied. The migration rate indicates the number of individuals that are migrated, while the migration frequency represents the number of iterations between two consecutive migrations. These two values are crucial for the coupling between the islands in the dGA. Table 9.9 summarizes the results using different combinations of these parameters. Upon examining the average fitness column (f), it is observed that a lower value of migration rate ($rate = 1$) is better than a higher value. A high coupling among islands (a low value of migration frequency) is not beneficial for this problem. The optimum value of migration frequency is 20, since if this value is further increased (resulting in a looser coupling among islands) the average fitness decreases.

Analysis of the Problem

From the previous analysis, it is concluded that the best settings for the problem instances of the fragment assembly problem is a population size of 512 individuals, with F1 as fitness function, OR as crossover operator (with probability 1.0), and with a swap mutation operator (with probability 0.3). The migration in dGAs occurs in a unidirectional ring manner, sending one single randomly chosen individual to the neighbor sub-population. The target population incorporates this individual only if it is better than its presently worst solution. The migration step is performed every 20 iterations in every island in an asynchronous way. A summary of the conditions for the experimentation is found in Table 9.10.

Table 9.7 Population Size Analysis

Popsize	Seq.		LAN					
	n = 1		n = 2		n = 4		n = 8	
	f	t	f	t	f	t	f	t
128	8773	1.7	53876	11	16634	3	23118	3.5
256	7424	0.01	76447	29	44846	7	21305	1.9
512	24023	56	86490	25	66854	14	56776	6.3
1024	21012	60	76263	30	60530	13	47026	7.1
2048	23732	67	54298	32	49049	14	32494	3.3

Table 9.8 Operator rate analysis (Pc-Pm)

P_c-P_m	Seq.			LAN								
	$n=1$			$n=2$			$n=4$			$n=8$		
	b	f	t	b	f	t	b	f	t	b	f	t
0.3-0.0	8842	7817	0	12e39	9145	0.2	11e00	8775	0.1	10284	7932	0.01
0.3-0.1	1e824	12223	33	91989	61e49	16	e1109	43822	8	34845	29e82	3.6
0.3-0.2	22e83	17e87	33	90891	70342	1e	70808	e9e81	8.1	e0338	41728	3.6
0.3-0.3	27466	20478	33	98341	77048	1e	78339	86137	7.8	e9242	e1174	3.6
0.5-0.0	8908	7820	0	29485	12981	0.5	13829	10238	0.2	12788	8822	0.03
0.5-0.1	18103	12800	45	83121	61355	20	54930	47894	11	40e23	31871	4.9
0.5-0.2	22706	19038	44	9e3e2	77e83	20	73333	62983	11	53328	45378	5
0.5-0.3	28489	23180	45	101172	84300	22	80102	70013	11	59948	53e87	5
0.7-0.0	9137	7459	0	28221	12540	0.8	14702	11099	0.2	18089	8938	0.05
0.7-0.1	17140	14068	56	86284	67225	27	59714	50899	14	39862	33719	8.3
0.7-0.2	28358	24023	56	98133	86490	25	7e824	66554	14	68021	56776	8.3
0.7-0.3	28359	25028	56	104641	84065	28	84732	71897	14	63482	53212	8.1
1.0-0.0	8892	7505	0	21864	14561	1.1	26294	12598	0.4	18732	9897	0.11
1.0-0.1	19485	16242	74	90815	71252	35	67087	55713	19	42850	33783	7.9
1.0-0.2	27564	22881	74	103231	81300	35	81417	71983	20	56871	50154	8.4
1.0-0.3	33071	27500	74	107148	88653	36	88389	74048	18	66588	58556	8.5

Table 9.9 Migration policy analysis (freq-rate)

freq-rate	LAN					
	$n = 2$		$n = 4$		$n = 8$	
	b	f	b	f	b	f
5–1	99904	76447	75317	62908	52127	35281
5–10	61910	37738	68703	55071	56987	52128
5–20	92927	72445	72029	66368	59473	54312
20–1	98133	86490	75824	66854	66021	56776
20–10	82619	45375	70497	57898	53941	48968
20–20	89211	74236	72170	65916	59324	53352
50–1	95670	70728	77024	65257	64612	55786
50–10	92678	41465	66046	51321	59013	51842
50–20	95374	76627	72540	62371	59923	52650

Table 9.11 shows the results and performance with all the data instances and algorithms described in this chapter. For two instances, it is clear that the distributed version outperforms the serial version. The distributed algorithm yields better fitness values and is faster than the sequential GA. Let us now go in deeper details on these claims.

For the first problem instance, the parallel GAs sampled fewer points in the search space than the serial one, while for the second instance the panmictic algorithm is mostly similar in the required effort with respect to the parallel ones.

Increasing of the number of islands (and CPUs) results in a reduction in search time, but it does not lead to a better fitness value. For the second problem instance, the average fitness was improved by a larger number of islands. However, for the first problem instance, it is observed a reduction in the fitness value as the number of CPUs are increased. This counterintuitive result clearly states that each instance has a different number of optimum number of islands from the point of view of the accuracy.

The best tradeoff is for two islands $(\mathrm{n} = 2)$ for the two instances, since this value yields a high fitness at an affordable cost and time.

Table 9.12 gives the speed-up results. As it can be seen in the table, almost linear speedup is obtained for the first problem instance. For the second instance also, there

Table 9.10 Parameters when heading and optimum solution of the problem

Independent runs	30
Popsize	512
Fitness function	F1
Crossover	OR (1.0)
Mutation	Swap (0.3)
Cutoff	30
Migration frequency	20
Migration rate	1

Table 9.11 Results of Both Problem Instances

		38524243_4				38524243_7			
		b	f	e	t	b	f	e	t
Sequential		33071	27500	810274	74	78624	67223	502167	120
LAN	$n=2$	107148	88653	733909	36	156969	116605	611694	85
	$n=4$	88389	74048	726830	18	158021	120234	577873	48
	$n=8$	66588	58556	539296	8.5	159654	119735	581979	27

is a good speedup with a low number of islands (two and four islands); eight islands make the efficiency decrease to a moderate speedup ().

Finally, Table 9.13 shows the global number of contigs computed in every case. This value is used as a high-level criterion to judge the whole quality of the results since, as mentioned before, it is difficult to capture the dynamics of the problem into a mathematical function. These values are computed by applying a final step of refinement with a greedy heuristic regularly used in this application. It is found that in some (extreme) cases it is possible that a solution with a better fitness than other one generates a larger number of contigs (worse solution). This is the reason for still needing research to get a more accurate mapping from fitness to contig number. The values of this table confirm again that all the parallel versions outperform the serial versions, thus advising the utilization of parallel GAs for this application in the future.

Conclusions

The DNA fragment assembly is a very complex problem in computational biology. Since the problem is NP-hard, the optimal solution is impossible to find for real cases, except for very small problem instances. Hence, computational techniques of affordable complexity such as heuristics are needed for this problem.

The sequential Genetic Algorithm used here solves the DNA fragment assembly problem by applying a set of genetic operators and parameter settings, but does take a large amount of time for problem instances that are over 15k base pairs. The distributed version has taken care of this shortcoming. The test data are over 77K base pairs long. The results obtained by the parallel algorithms are encouraging not

Table 9.12 Speed-up

		38524243_4			38524243_7		
		n CPUs	1 CPU	Speedup	*n* CPUs	1 CPU	Speedup
LAN	$n=2$	36.21	72.07	1.99	85.37	160.15	1.87
	$n=4$	18.32	72.13	3.93	47.78	168.20	3.52
	$n=8$	8.52	64.41	7.56	26.81	172.13	6.42

Table 9.13 Final Best Contigs

		38524243_4	38524243_7
Sequential		5	4
	n = 2	3	2
LAN	n = 4	4	1
	n = 8	4	2

only because of their low waiting times, but also because of their high accuracy in computing solutions of even just 1 contig. This is noticeable since it is far from triviality to compute optimal solutions for real-world instances of this problem.

To curb the problem of premature convergence for example, a restart technique can be employed in the islands. Another interesting point of research would be to incorporate different algorithms in the islands, such as greedy or simulated annealing, and to study the effects this could have on the observed performance.

9.3 Investigating Parallel Genetic Algorithms on Job Shop Scheduling Problems

9.3.1 Introduction

Job shop scheduling problems (JSSP's) are computationally complex problems. Because JSSP's are NP-hard – i.e., they can't be solved within polynomial time -brute-force or undirected search methods are not typically feasible, at least for problems of any size. Thus JSSP's tend to be solved using a combination of search and heuristics to get optimal or near optimal solutions. Among various search methodologies used for JSSPs, the Genetic Algorithm (GA), inspired by the process of Darwinian evolution, has been recognized as a general search strategy and optimization method which is often useful in attacking combinatorial problems. Since Davis proposed the first GA-based technique to solve scheduling problems in 1985, GAs have been used with increasing frequency to solve JSSP's. In contrast to local search techniques such as simulated annealing and tabu-search, which are based on manipulating one feasible solution, the GA utilizes a population of solutions in its search, giving it more resistance to premature convergence on local minima. The main difficulty in applying GAs to highly constrained and combinatorial optimization problems such as JSSP's is maintaining the validity of the solutions. This problem is typically solved by modifying the breeding operators or providing penalties on infeasible solutions in the fitness function. Although resistant to premature convergence, GAs are not immune. One approach to reduce the premature convergence of a GA is parallelization of the GA into disjoint subpopulations, which is also a more realistic model of nature than a single population. Currently, there are two kinds of parallel

GAs (PGAs) that are widely used: coarse-grain GAs (cgGAs) and fine-grain GAs (fgGAs). Both will be studied in the context of JSSP's.

9.3.2 Job Shop Scheduling Problem

Job shop scheduling, in general, contains a set of concurrent and conflicting goals to be satisfied using a finite set of resources. The resources are called machines and the basic tasks are called jobs. Each job is a request for scheduling a set of operations according to a process plan (or referred to as process routing) which specifies the precedence restrictions. The main constraint on jobs and machines is that one machine can process only one operation at a time and operations cannot be interrupted. Usually the general JSSP is denoted as *n*x*m*, where *n* is the number of jobs and *m* is the number of machines. The operation of job *i* on machine *j* is denoted by operation $(i,\ j)$. The problem is to minimize some performance criterion. This section discusses the most widely used criterion, i.e., the time to completion of the last job to leave the system – the makespan.

One useful model used to describe JSSP's is the disjunctive graph, $G = (N, A, E)$, where *N* is the node set, *A* is the conjunctive arc set, and *E* is the disjunctive arc set. The nodes *N* correspond to all of the operations and two dummy nodes, a source and a sink. The conjunctive arcs *A* represent the precedence relationships between the operations of a single job. The disjunctive arcs *E* represent all pairs of operations to be performed on the same machine. All arcs emanating from a node have the processing time of the operation performed at that node as their length. The source has conjunctive arcs with length zero emanating to all the first operations of the job and the sink has the conjunctive arcs.

A feasible schedule corresponds to a selection of exactly one arc from each disjunctive arc pair such that the resulting directed graph is acyclic. The problem of minimizing the makespan reduces to finding a set of disjunctive arcs, which minimize the length of the longest path or the critical path in the directed graph.

In JSSP's, two classes of schedules are defined. The first is *semi-active* schedules, the other is *active* schedules. Semi-active schedules are feasible schedules in which no operation can be completed earlier without changing the job sequence on any of the machines. Active schedules are feasible schedules in which no operation can be completed earlier by changing the processing sequence on any of the machines without delaying some other operation. Clearly the set of active schedules is a subset of the set of semi-active schedules and optimal schedules are active schedules. Thus, in optimizing makespan, it is sufficient to consider only active schedules. A systematic approach to generate active schedules was proposed by Giffler and Thompson. Because this procedure is closely related to the genetic operators, a brief outline of the G&T algorithm is given in Figure 9.10. The key condition in the G&T algorithm is the inequality $r_{ij^*} < t(C)$ in Step 3, which generates a conflict set consisting only of operations competing for the same machine. Once one operation is decided, it is impossible to add any operation that will complete prior to $t(C)$, making the generated schedule an active schedule.

9.3.3 Genetic Representation and Specific Operators

"Classical" GAs use a binary string to represent a potential solution to a problem. Such a representation is not naturally suited for ordering problems such as the Traveling Salesperson Problem (TSP) and the JSSP, because no direct and efficient way has been found to map possible solutions 1:1 onto binary strings. Two different approaches have been used to deal with the problem of representation. The first is an *indirect* representation, which encodes the instructions to a schedule *builder*. Some examples of an indirect representation are job order permutation and prioritization of scheduling rules. In these schemes, the schedule builder guarantees the validity of the schedules produced. Another approach is to use a *direct* representation which encodes the *schedule itself*. Some examples of direct representations use encodings of the operation completion times or the operation starting times. In such a representation, not every encoding corresponds to a valid schedule. If invalid encodings are allowed in the population, repair methods or penalty functions are required to maintain the validity of the schedules. However, use of penalty functions is inefficient for JSSP's because the space of valid schedules is very small compared to the space of possible schedules. Thus, the GA will waste most of its time on invalid solutions. Another problem with a "classical" GA representation is that simple crossover or mutation on strings nearly always produces infeasible solutions. Previous researchers used some variations on standard genetic operators to address this problem. Well-known examples are sequencing operators devised for the TSP.

This approach uses a direct representation, which encodes the operation starting times. The number of the fields on the chromosome is the number of operations. The genetic operators are inspired by the G&T algorithm. Some related

Step 1:
Let $\boldsymbol{C}$ contain the first schedulable operation of each job;
Let $r_{ij} = 0$, for all operations $(i,\ j)$ in $\boldsymbol{C}$.
(r_{ij} is the earliest time at which operation $(i,\ j)$ can start.)
Step 2:

$$\text{Compute } t(\boldsymbol{C}) = \min_{(i,j\in\boldsymbol{C})} \{r_{ij} + p_{ij}\}$$

and let j^* denote the machine on which the minimum is achieved.
(p_{ij} is the processing time of operation (i, j))
Step 3:
Let $\boldsymbol{G}$ denote the conflict set of all operations $(i,\ j^*)$ on machine j^* such that

$$r_{ij^* < t(C)}$$

Step 4:
Randomly select one operation from $\boldsymbol{G}$.
Step 5:
Delete the operation from $\boldsymbol{C}$; include its immediate successor in $\boldsymbol{C}$, update r_{ij} in $\boldsymbol{C}$ and return to step 2 until all operations are scheduled.

Fig. 9.10 The Giffler and Thompson algorithm coming from all the last operations

Table 9.14 G&T-algorithm-based GA approaches

Reference	Representation	Crossover
Yamada and Nakano (1992) [18]	completion time	uniform
Storer et al. (1992) [15]	perturbed processing time	standard
Dorndorf and Pesch (1993) [3]	starting time	standard
Dorndorf and Pesch (1995) [4]	priority rule	standard
Kobayashi et al. (1995) [9]	job order	subsequence exchange
Lin et al.	starting time	time horizon exchange

approaches which are G&T-algorithm-based are briefly reviewed. Yamada and Nakano used operation completion times for their representation. They proposed using GA/GT crossover, which ensures assembling valid and active schedules. The GA/GT crossover works as follows: at each decision point in the G&T algorithm (step 4 in Fig 9.10), one parent is selected randomly. However, in GA/GT crossover, the schedulable operation which has the earliest completion time reported in the parental schedule is chosen to be scheduled next. The effect of GA/GT crossover is the same as applying uniform crossover and using the G&T algorithm to interpret the resulting invalid chromosomes. Dorndorf and Pesch encode the operation starting times and apply the G&T algorithm to decode the invalid offspring which are generated from standard crossover. Other approaches are similar to the two above. The G&T algorithm is used as an interpreter to decode any offspring into an active schedule. Table 9.14 lists the G&T-algorithm-based GA approaches. These approaches are designed to transmit "useful characteristics" from parents for the creation of potentially better offspring. These "useful characteristics" can be priority rules or job sequences, depending on the representation and crossover methods used. In JSSPs, the temporal relationships among all operations in a schedule are important. Simply working on the chromosome level usually focuses on only a small part of the schedule and overlooks the change of the temporal relationships in the whole schedule. In contrast to previous approaches, which work on the chromosome level, the time horizon exchange (THX) crossover, which works on the *schedule* level is designed. THX crossover randomly selects a crossover point just like a standard crossover, but instead of using the crossover point to exchange two chromosomes, THX crossover uses the crossover point as a scheduling decision point in G&T algorithm to exchange information between two schedules. Figure 9.11 shows an example of THX crossover in Fischer and Thompson's (FT) 6×6 problem. The portion of the child schedule before the crossover point is exactly the same as in one parent. The temporal relationships among operations in the remaining portion are inherited from the other parent to the extent possible (i.e., while maintaining a valid schedule).

Another important operator in GAs is mutation. The THX mutation operator is based on the disjunctive graph of the schedule. Although exchanging a single pair of adjacent tasks which are on the same machine and belong to a critical path can preserve the acyclic property of the directed graph, the number of child schedules that are better than the parent tends to be very limited, as was observed by Grabowski

et al. They defined a *block* as a sequence of successive operations on the critical path which are on the same machine with at least two operations. The reversal of a critical arc can only lead to an improvement if at least one of the reversed operations is either the first or the parent 1 last operation of the block. Thus this mutation focuses on the block. Two operations in the block are randomly selected and reversed. After the child is generated, the G&T algorithm is applied to interpret the child. Thus, no cycle detection is needed. Furthermore, the G&T algorithm guarantees that the two selected operations are reversed in the new schedule and that the new schedule is active.

9.3.4 Parallel Genetic Algorithms for JSSP

Although "classical" GAs can be made somewhat resistant to premature convergence (i.e., inability to search beyond local minima), there are methods, which can be used to make GAs even more resistant. PGAs retard premature convergence by maintaining multiple, somewhat separated subpopulations which may be allowed to

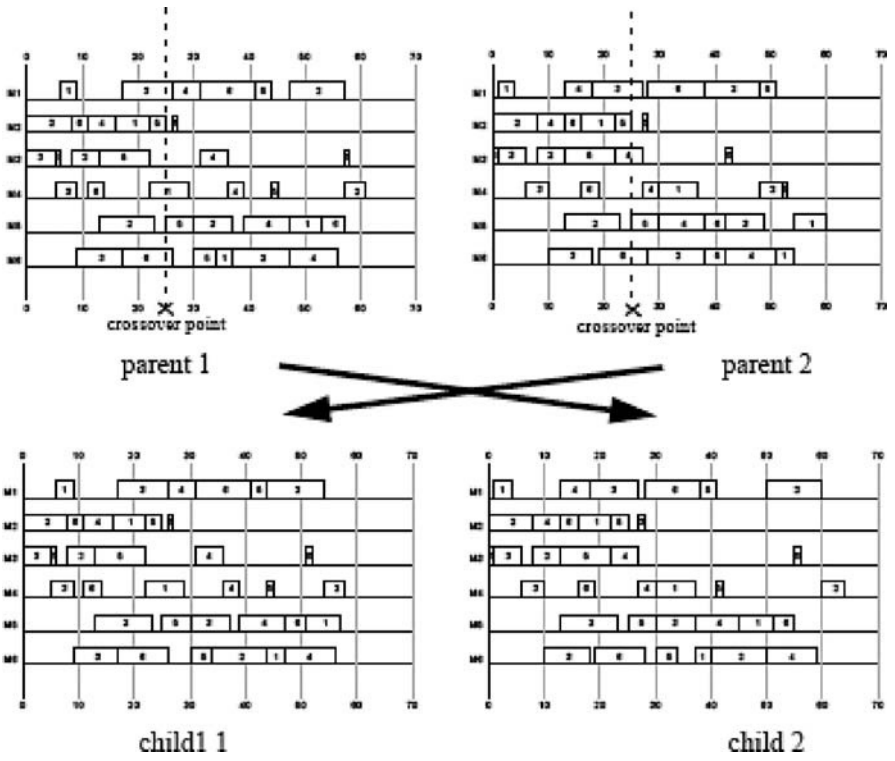


Fig. 9.11 An example of THX crossover

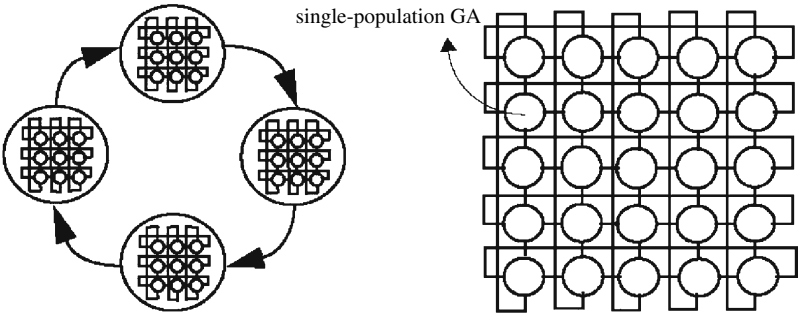


Fig. 9.12 Examples of the hybrid models

evolve more independently (or, more precisely, by employing non-panmictic mating). Two fundamental models of PGAs can be distinguished in the literature. The first is *fine-grained* GAs (fgGAs), in which individuals are spatially arrayed in some manner and an individual in the population can interact only with individuals "close" to it. The topology of individuals in the template defining the breeding "neighborhood" determines the degree of isolation from other individuals and therefore strongly influences the diversity of the individuals in the population. All the individuals can be considered to be continuously moving around within their neighborhoods, so that global communication is possible, but not instantaneous.

The second is the *coarse-grained* GAs (cgGAs), also called island-parallel GAs, in which each node is a subpopulation performing a single-population GA. At certain intervals, some individuals may migrate from one subpopulation to another. The rate at which individuals can migrate globally is typically much smaller than found in fgGAs. In this application, a two-dimensional torus is used as the neighborhood topology to study the fgGA model and an island GA connected in a ring to study the cgGA model.

Two hybrid models are also proposed in this section. One is an embedding of fgGAs into cgGAs. Fig 9.12(a) shows an example in which each subpopulation on the ring is a torus. The frequency of migration on the ring is much smaller than that within the torus. The other hybrid model is a "compromise" between a cgGA and a fgGA – the connection topology used in the cgGA is one which is typically found in fgGAs, and a relatively large number of nodes is used. Fig 9.12(b) shows an example in which each node of the torus is a single-population GA. The frequency of migration resembles that typically found in cgGAs. The performance of the test models with various population sizes is dipicted in Figure 9.13.

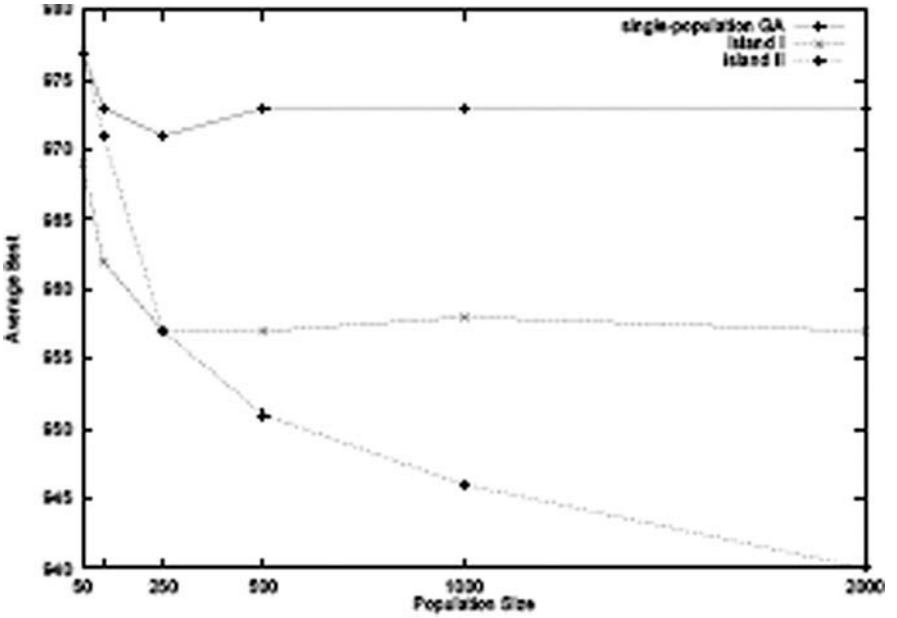


Fig. 9.13 Avg. (100 runs) best indiv. for three test models, various population sizes

9.3.5 Computational Results

The configurations described have been implemented in GALOPPS, a free-ware GA development system from the MSU GARAGe, and run on a Sun Ultra 1. As a benchmark, two FT problems, FT10×10 and FT20×5, were tested. These two FT problems are of particular interest because almost all JSSP algorithms proposed have used them as benchmarks. Table 9.15 summarizes the best results obtained by previous approaches for the two FT problems. Except for the first three approaches, which are based on branch and bound methods, the remaining approaches are GA-based methods. By using single-population GAs with the THX operators, the global optima were found for the two problems, which are 930 and 1165, respectively. The FT10×10 was also used to evaluate the effectiveness of PGAs and to compare the performance of the various PGA models in the following subsections.

The Effect of Parallelizing GAs

To investigate the effect of parallelizing GAs, one single-population GA and two cgGAs with different population sizes were used on the FT10×10 problem. In all runs, the crossover and mutation rates were 0.6 and 0.1 respectively, and offspring replaced their parents, with elitism protecting the best individual from replacement. The total population size is varied in each case to test for its effect. The population

Table 9.15 Best results obtained by previous approaches on the two FT problems

Reference	10×10	20×5
Baker and McMahon (1985)[22]	960	1303
Adams *et al*. (1988)[20]	930	1178
Carlier and Pinson (1989)[21]	930	1165
Nakano and Yamada (1991)[14]	965	1215
Yamada and Nakano (1992)[18]	930	1184
Storer *et al*. (1993)[15]	954	1180
Dorndorf and Pesch (1993)[3]	930	1165
Fang *et al*. (1993)[5]	949	1189
Juels and Wattenberg (1994)[8]	937	1174
Mattfeld *et al*. (1994)[12]	930	1165
Dorndorf and Pesch (1995)[4]	938	1178
Bierwirth (1995)[1]	936	1181
Kobayashi *et al*. (1995)[9]	930	1173
Lin *et al*.	930	1165

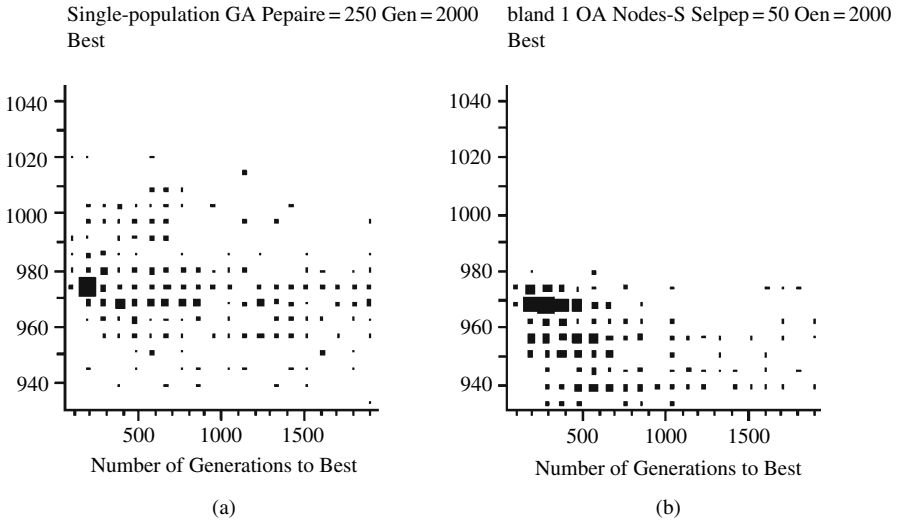


Fig. 9.14 (a) the single-population GA (b) the island I GA

sizes used was 50, 100, 250, 500, 1000, and 2000. Both cgGAs, called island I and island II, are connected in a one-way ring. The best individual is migrated to the next neighbor every 50 generations. The number of nodes in the island I GA was fixed at 5, so the subpopulation size is the total size divided by 5. In the island II GA, the subpopulation size is fixed at 50, so the number of nodes is obtained by dividing the total population size by 50.

Fig 9.14 shows the average best results of the three models on various population sizes based on 100 runs. The single-population GA doesn't show any improvement in performance after the population size 250 mark. The reason is that the single-population GA cannot maintain the diversity in the population as well as the PGA

Table 9.16 The population structures of the PGA models

Popsize	Island I	Island II	Torus	Hybrid I	Hybrid II
250	50:5	50:5	$2:25\times5$	2:5 islands, each island: 5×5 torus	$10:5\times5$
500	100:5	50:10	$2:25\times10$	2:5 islands, each island: 10×5 torus	$10:5\times10$
1000	200:5	50:20	$2:25\times20$	2:5 islands, each island: 10×10 torus	$10:10\times10$
2000	400:5	50:40	$2:25\times40$	2:5 islands, each island: 10×20 torus	$20:10\times10$

approaches. This loss of diversity causes premature convergence. The problem also appears in the island I GA model. Although premature convergence strongly deters further improvement after the population size 250 mark, the island I GA still outperforms the single-population GA. The island II GA doesn't suffer as much from premature convergence. The larger the number of subpopulations, the better diversity is maintained. An average best of 940 is reached when the population size is 2000. By considering the average turnaround time of each node to calculate a fixed number of generations, the speed-up of PGAs can be analyzed. In general, increasing the number of processors leads to approximately linear speed-up. For example, for a total population size fixed at 1000, the speed-up for numbers of nodes set to 5 and 20 are 4.7 and 18.5, respectively. The degraded performance is due to the communication overhead. In PGAs, the time needed to reach a given solution quality is crucial. Figure 9.14 shows the two-dimensional cell plot of the single-population GA and the island I GA with population size 250, based on 1000 runs. In this figure, it is observed that the distribution of the results moves to the left corner in the island I GA. That is, the parallelization of the GA yields better results using fewer evaluations. Actually, the average number of generations to obtain the best result in the island I GA is 732, compared to 852 for the single-population GA. Because the average best result of the island I GA is better than that of the single-population GA, the speed-up under "time-to-solution" is surely >5.8.

9.3.6 Comparison of PGA Models

Five PGA schemes – the two cgGAs discussed in 9.3.1, plus one fgGA torus model and two hybrid models are examined. The migration interval in cgGAs is 50 generations (i.e. an exchange between subpopulations every 50 generations). The population structures are shown in Table 9.16 in a subpopulation_size:connection_topology format. In the torus model, the subpopulation size is fixed at 2. In the hybrid I model, each island on the ring is a torus and the number of islands is fixed at 5.

Figure 9.15 shows the average best of the five PGA models based on 100 runs. The hybrid I and torus models have similar performance because both models are based on the fgGA model. Although both models are inferior to island I when the population size is less than 1000, their average best result improved for larger population sizes. The island II and hybrid II models are superior to the other approaches. The essential island structure of both models successfully alleviates premature

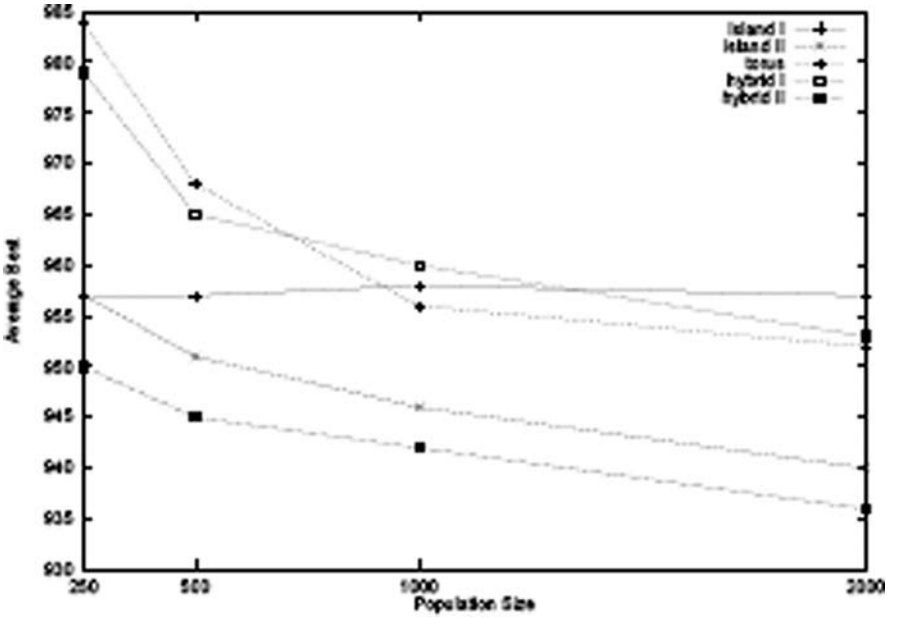


Fig. 9.15 Average best of the five PGA models with various population size

convergence. The connection topology of fgGAs in the hybrid II model supports the diffusion of genetic material to different subpopulations and further enhances its search ability. Thus the excellent results of the hybrid II model are achieved by combining the merits found in cgGAs and fgGAs. Notice that in the hybrid II model at population size 2000, the optimal schedule is found 40 times in 100 runs. The *average* result is 936, which is within 0.7% of the optimum, and the standard deviation is 5.62. The superiority of this method with the best PGA model is retained at significance levels better than 0.0001 compared with the results of Juels and Wattenberg and the results of Mattfeld *et al*.

In this test problem, fgGAs appear to lose genetic diversity too quickly, in comparison to cgGAs. Improvement can be made if a different migration strategy is applied. In cgGAs, increasing the number of islands improves performance more than simply increasing the total population size. Additionally, a good connection topology can further increase the performance. Best results were obtained with the hybrid model consisting of cgGAs connected in a fgGA-style topology.

9.4 Parallel Genetic Algorithm for Graph Coloring Problem

9.4.1 Introduction

Graph coloring problem (GCP) belongs to the class of NP-hard combinatorial optimizations problems. GCP is defined for an undirected graph as an assignment of available colors to graph vertices providing that adjacent vertices are assigned different colors and the number of colors is minimal. There are many variants of the problem when some additional assumptions are made. Intensive research conducted in this area resulted in a large number of exact and approximate algorithms, heuristics and metaheuristics. GCP was the subject of Second DIMACS Implementation Challenge in 1993 and Computational Symposium on Graph Coloring and Generalizations in 2002. Genetic algorithms (GA) are metaheuristics often used for GCP. Recently a number of parallel versions of GA were studied. This approach is based on co-evolution of a number of populations that exchange genetic information during the evolution process according to a communication pattern. In this section results of the experiments with parallel genetic algorithms (PGA) for graph coloring problem are presented. Two new recombination operators for coloring chromosomes are used: SPPX (Sum -Product Partition Crossover) in which simple set operations and random mechanisms are implemented, and CEX (Conflict Elimination Crossover) that is focused on the offspring quality but much more complex. In computer simulations of PGA, DIMACS benchmarks are used. The obtained results are very promising and encourage future research focused on PGA and new genetic operators for graph coloring problems.

Migration Model of Parallel Genetic Algorithm

There are many model of parallelism in evolutionary algorithms: master-slave PGA, migration based PGA, diffusion based PGA, PGA with overlapping subpopulations, population learning algorithm, hybrid models etc. Migration models of PGAs consist of a finite number of subpopulations that evolve in parallel on their "islands" and only occasionally exchange the genetic information under control of a migration operator. Co-evolving subpopulations are built of individuals of the same type and are ruled by one adaptation function. The selection process is decentralized. In this model the migration is performed on a regular basis. During the migration phase every island sends its representatives (emigrants) to all other islands and receives their representatives (immigrants) from all co-evolving subpopulations. This topology of migration reflects so called "pure" island model. The migration process is fully characterized by migration size, distance between populations and migration scheme. Migration size determines the emigrant fraction of each population. The distance between migrations determines how often the migration phase of the

algorithm occurs. Two migration schemes are applied: migration of best individuals of the subpopulation or migration of individuals randomly selected.

In this algorithm a specific model of migration in which islands use two copies of genetic information is used: migrating individuals still remain members of their original subpopulation. In other words they receive new "citizenship" without losing the former one. Incoming individuals replace the chromosomes of host subpopulation at random. Then, a selection process is performed. The rationale behind such a model is as follows. Even if the best chromosomes of host subpopulation are eliminated they shall survive on other islands where their copies were sent. On the other hand any eliticist scheme or preselection applied to the replacement phase leads to premature elimination of worse individuals and lowers the overall diversity of subpopulation.

9.4.2 Genetic Operators for GCP

In graph coloring problem *k*-colorings of graph vertices are encoded in chromosomes representing set partitions with exactly k blocks. In partition representation each block of partition does correspond to a single color. In assignment representation available colors are assigned to an ordered sequence of graph vertices. In this section a collection of genetic crossover, mutation and selection operators used in PGA are explained.

Sum-product Partition Crossover

The first recombination operator called Sum-Product Partition Crossover (SPPX) employs for offspring generation simple set sum and set product operations on block of partitions and a random mechanism of operand selection from randomly determined 4 parental chromosomes.

SPPX is composed of two procedures PRODUCT and SUM which are applied to the pair of chromosomes $\mathrm{p} = \{V_1^p, \ldots, V_k^p\}$, $\mathrm{r} = \{V_1^r, \ldots, V_k^r\}$ and produce a pair of chromosomes $\mathrm{s} = \{V_1^s, \ldots, V_m^s\}$, $\mathrm{t} = \{V_1^t, \ldots, V_n^t\}$ with probabilities of elementary operations satisfying:

$0 \leq \mathrm{Prob(PRODUCT)} < \mathrm{Prob(SUM)} \leq 1$. A pseudocode of the procedure SPPX is presented in Figure 9.16.

Illustration I

Four parents represent different 3{colorings of a graph with 10 vertices: p1={ABC,DEFG, HIJ}, r1={CDEG,AFI,BHJ}, p2={CDG,BEHJ,AFI} and

r2={ACGH,BDFI, EJ}. Assume Prob (PRODUCT) = 0.5, Prob(SUM)=0.7. Let rand1=0.4 and PRODUCT is computed with h = 3, j = 2. Thus, $V_3{}^p = \{HIJ\}$ and $V_2{}^r = \{AFI\}$. Then $V_1{}^s = V_1{}^t = \{I\}$ and s1 = {I, ABC, DEFG, HJ} and t1 = {I, CDEG, AF, BHJ}. Let rand2=0.3 and SUM is computed with h = 2, j = 1. Thus, $V_2{}^p = \{BEHJ\}$ and $V_1{}^p = \{ACGH\}$. This leads to $V_1{}^s = V_1{}^t = \{ABCEGHJ\}$ and then s2 = {ABCEGHJ, D, FI} and t2 = {ABCEGHJ, DFI}. As a result of the crossover there are four children: s1, t1, s2 and t2 representing 2, 3 and 4 colorings of the given graph.

Notice that operation PRODUCT may increase the initial number of colors while the operation SUM may reduce this number. The probability of PRODUCT should be lower then the probability of SUM. The recombination operator SPPX that can be used as a versatile operator in evolutionary algorithms for many other partition problems.

```
procedure: SPPX (p1,p2,r1,r2,s1,s2,t1,t2,Prob(PRODUCT)
begin
  s1=t1=s2=t2=∅;
  generate random numbers rand1, rand2 : 0 ≤rand1,ran
  if rand1 ≤ Prob(PRODUCT) then PRODUCT(p1,r1,s1,t1);
  if rand2 ≤ Prob(SUM) then SUM(p2,r2,s2,t2);
end SPPX;
PRODUCT(p,r,s,t)
begin
  select random h (1≤h≤k) and j (1≤j≤l);
  V_1^s = V_1^t = (V_h^p ∩ V_j^r);
  for i = 1 to k do
    if i ≠ h do if (V_i^p \ V_1^s) nonempty then
      add next block V_i^p \ V_1^s to s;
  for i = 1 to l do
    if i ≠ j do if (V_i^r \ V_1^t) nonempty then
      add next block V_i^r \ V_1^t to t;
end PRODUCT;
SUM (p,r,s,t)
begin
  select random h (1≤h≤k) and j (1≤j≤l);
  V_1^s = V_1^t = (V_h^p ∪ V_j^r);
  for i = 1 to k do
    if i ≠ h do if (V_i^p \ V_j^r) nonempty then
      add next block V_i^p \ V_j^r to s;
  for i = 1 to l do
    if i ≠ j do if (V_i^r \ V_h^p) nonempty then
      add next block V_i^r \ V_h^p to t;
end SUM;
```

Fig. 9.16 The recombination operator SPPX

Conflict Elimination Crossover

In conflict -based crossovers for GCP an assignment representation of colorings are used and the offspring try to copy conflict -free colors from their parents. The next recombination operator called Conflict Elimination Crossover (CEX) reveals some similarity to the classical crossover. Each parental chromosome is partitioned into two blocks. The first block consists of conflict free nodes while the second block is built of the remaining nodes that break the coloring rules.

The last block in both chromosomes is then replaced by corresponding colors taken from the other parent. This recombination scheme provides inheritance of all good properties of one parent and gives the second parent a chance to reduce the number of existing conflicts. However, if a chromosome represents a feasible coloring the recombination mechanism is not working. Therefore, the recombination must be combined with an efficient mutation mechanism. The operator CEX shown in Figure 9.17 is almost as simple and easy to implement as the classical crossover

Illustration 2

Two parents represent different 5-colorings of a graph with 10 vertices i.e. sequences: p =<5,2,3,1,1,4,3,5,1,2> and r =<1,4,5,2,3,3,2,4,2,1>. Vertices with assigned wrong colors are marked by bold fonts.

Replacing the vertices with color conflicts by vertices taken from the other parent the following two chromosomes: s =<5,2,5,2,1,3,3,5,1,1> and t =<1,4,3,2,3,3,3, 4,2,2> are obtained .It can be observed that obtained chromosomes represent now two different 4-colorings of the given graph (reduction by 1 with respect to initial coloring) and the number of color conflicts is now reduced to 2 in each chromosome.

Union Independent Set Crossover

The greedy operator proposed by Dorne and Hao called Union Independent Sets (UISX) works on pairs of independent sets taken from two parent colorings. In any feasible graph coloring all graph vertices are partitioned into blocks that are disjoint independent sets (IS). A coloring is not feasible if it contains at least one block,

procedure: CEX (p,r,s,t)
begin
s = r;
t = p;
copy conflict-free vertices V_{CF}^{P} from p to s;

copy conflict-free vertices V_{CF}^{R} from r to t;
end

Fig. 9.17 The recombination operator CEX

```
procedure: GPX (p0,p1,s)
begin
s = ϕ, i = 1;
repeat
select block V with maximum cardinality from the partition p(i mod2);
s = s ∪V—add the block V to partition s;
remove all vertices of V from p0 and p1;
i = i + 1;
until (i > k) or (all blocks of p1 and p2 empty);
assign randomly all unassigned vertices to blocks of s;
end
```

Fig. 9.18 The modified recombination operator GPX

which is a non-independent set. Each block of a partition is assigned one color. "If we try to maximize the size of each IS by a combination mechanism, we will reduce the sizes of non-independent sets, which in turn helps to push these sets into independent sets". In the initial step disjoint ISs in both parents are determined. Then coloring for the first child is computed. First, select the maximum IS from the first parent and compute set intersections with ISs from the second parent. The union of a pair of ISs with maximum intersection is colored in the offspring with the IS color from the first parent. In the case of a tie a random IS is always chosen. Then the colored vertices are removed from the both parents and the coloring procedure is repeated as long as possible. The vertices without any color are assigned the original color from the first parent. The coloring for the second child is computed with reversed roles of both parents.

Greedy Partition Crossover

The method called Greedy Partition Crossover (GPX) was designed by Galinier and Hao for recombination of colorings or partial colorings in partition representation. It is assumed that both parents are randomly selected partitions with exactly k blocks that are independent sets. The result is a single offspring (a coloring or partial coloring) that is built successively in a greedy way. In each odd step the maximum block is selected from the first parent. Then, the block is added to the result and all its nodes are removed from the both parents. In each even step the maximum block is selected from the second parent. Then, add the block to the result and remove all its nodes from the both parents. The procedure is repeated at most k times since, in some cases, the offspring has less blocks than the parents (see Example 3). Finally, unassigned vertices (if they exist) are assigned at random to existing blocks of partition. A corrected version of GPX is shown in Figure 9.18.

The first parent is replaced by the offspring while the second parent returns to population and can be recombined again in the same generation. GPX crossover is performed with a constant probability.

Illustration 3

Two parents represent different 3{colorings of a graph with 10 vertices, i.e. partitions: p0={ABFGI,CDE,HJ}, p1={ABF,CDEGHJ,I}.

For i = 1 the maximum block {CDEGHJ} is selected from p1 and is added to s. After removing the block vertices from the parents p0={ABFI}, p1={ABF, I}are obtained. For i = 2 the maximum block {ABFI} is selected from p0 and is added to s. Termination condition is satisfied and the result partition s={ABFI, CDEGHJ}, which is a valid 2-coloring is obtained.

Mutation Operators

Transposition is a classical type of mutation that exchanges colors of two randomly selected vertices in the assignment representation. The second mutation operation called First Fit is designed for colorings in partition representation and is well suited for GCP. In the mutation First Fit one block of the partition is selected at random and a conflict -free assignment of its vertices to other blocks using the heuristic is tried. First Fit. Vertices with no conflict-free assignment remain in the original block. Thus, as a result of the First Fit mutation the color assignment is partially rearranged and the number of partition blocks is often reduced by one.

Selection Operator

The quality of a solution is measured by the following cost function:

$$f(p) = \sum(u,v) \in E^{q(u,v)+d+c}, \text{ Where}$$

p is a graph coloring,
q is a penalty function for pairs of vertices connected by an edge (u,v) ε E:
q (u; v) = 2 when c(u) = c(v), and q(u; v) = 0, otherwise,
d { is a general penalty function applied to graph colorings:
d = 1, when $\sum(u,v) \in E^{q(u,v)} > 0$ and d = 0 when $\sum(u,v) \in E^{q(u,v)} = 0$,
C is the number of colors used.
The proportional selection is performed in this PGA with the fitness function 1/f (p).

9.4.3 Experimental Verification

For computer experiments graph coloring instances available in the web archive http://mat.gsia.cmu.edu/COLOR/instances.html were used. This is a collection of graphs in DIMACS format with known parameters, including graph chromatic numbers. In this computer program PGA for GCP two basic models of PGA: migration and master {slave can be simulated. It is possible to set up most parameters of

evolution, monitor evolution process on each island and measure both the number of generations and time of computations. In the preprocessing phase list of edges representation were converted into adjacency matrix representation. The program generates detailed reports and basic statistics.

The influence of migration scheme on the PGA efficiency is measured by the number of generations n needed to obtain an optimal coloring when the chromatic number is known. Experiments were performed on 5 graphs with the following parameters: population size = 60, number of islands = 5, migration rate = 5, crossover = SPPX, mutation = First Fit, and mutation probability = 0.1. All experiments were repeated 30 times. For all graphs migration of best individuals always gives the best results. Migration of random individuals is almost useless except huge graphs like mulsol.i.4 where random migration is also efficient. The experiments confirmed that the mutation First Fit is superior to the Transposition mutation for graph coloring problems. It works particularly well with CEX crossover.

In the main experiment the efficiency of all 4 crossover operators was tested.

Experiments were performed on 6 graphs with the following parameters:

population size = 60, number of islands = 3, migration rate = 5, migration size = 5, mutation = First Fit, and mutation probability = 0.1. All experiments were repeated 30 times. The results are presented in Table 9.17. All computer experiments were performed on a computer with AMD Athlon 1700+ processor (1472 MHz) and 256 MB RAM.

It is observed that the proposed crossover operators are efficient in terms of computation time. SPPX requires more generations then GPX in order to find an optimal coloring but it is simpler and therefore a bit faster then GPX. In some cases the operator UISX requires less generations then GPX but it always produces an optimal solution faster then two previous operators. The most efficient operator in the experiment is CEX, which dominates all other operators under the both criteria (except the smallest graph instance).

Table 9.17 Performance of the migration-based PGA with various crossover operators

No	Graph	Vertices	Edges	Colors	Crossover operator							
					UISX		GPX		SPPX		CEX	
					n	t[s]	n	t[s]	n	t[s]	n	t[s]
1	anna	1.18	193	11	19	2.0	21	6.0	61	3.2	15	1.3
2	david	87	106	11	22	2.0	24	3.9	74	3.6	20	1.0
3	huck	74	301	11	8	0.8	7	1.0	29	0.5	12	0.7
4	miles500	128	1170	20	95	9.5	59	40	152	38	100	3.0
5	mycie17	191	2360	8	18	2.0	21	7.9	76	7.6	20	1.0
6	mulsol_1	197	3925	49	90	31	60	35	180	34	58	2.0

9.4.4 Conclusion

In this application it is proved by computer simulation that parallel genetic algorithms can be efficiently used for the class of graph coloring problems. In island model of PGA the searched space is significantly enlarged and the migration between co-evolving subpopulations improves the overall convergence of the algorithm. PGA is particularly efficient for large scale problems like GCP. The results presented in this application encourage further research in this area. One obvious direction is to extend the experiments on other DIMACS benchmarks including a class of random graphs. It is also worth to consider some variants of SPPX operator that will make it more problem-oriented. The search for new efficient genetic operators for GCP still remains an open question.

9.5 Robust and Distributed Genetic Algorithm for Ordering Problems

9.5.1 Introduction

A distributed system can be described as a set of computers interconnected by communication links. The major advantages of a distributed system are resource sharing, flexibility and reliability. Due to the increasing availability of inexpensive personal computers and workstations at universities and industries, the use of distributed computing systems has also been growing rapidly. The concepts of distributed processing can be applied to a number of applications to enhance their performance. In this application, it is demonstrated that a simple crossover scheme used in conjunction with Distributed GA, can provide a simple and efficient means to solve ordering type of problems.

The basic strength and generality of GA lies in the fact, that if a problem can be expressed as a binary string, such that the string contains all the required information, then a search for the solution can be carried out successfully. However, there is a large class of problems, which do not easily map to binary strings. These include ordering or permutation problems. Hence, most ordering problems use an integer representation. The main difficulty encountered when using non-standard representation is the choice of a suitable crossover scheme. Different operators like PMX, OX, CX etc have been used in the past for ordering problems. But the major problem with most of the existing crossover operators is their high computational complexity. There have been several studies on parallel versions of Genetic Algorithms in the past. Many of these studies focus on achieving a speed-up in terms of execution time over the traditional GA. The speed-up is achieved by employing various techniques like computation of the objective function of all the chromosomes in parallel, division of the population into several sub populations, either to execute them in parallel or to restrict the selection and reproduction criterion . But

their major disadvantage is that they are problem specific and hence, limited in their scope. Moreover, these schemes require the use of parallel computers, which are expensive and not easily available. This distribution scheme focuses on improving the quality and consistency of the solution obtained by GA. Most importantly, the solution obtained in this distributed GA implementation is independent of the values of P_e, and P_m, used. Moreover, this scheme is generic in nature and is independent of the problem solved. Also, implementation of this scheme does not require any specialized hardware and it can be implemented easily on any existing network of computers. The basic concept in this distribution scheme is based on the use of several independent GA processes executing in parallel and cooperating with each other to produce a high-quality and consistent solution. The distributed environment used for Distributed GA implementation is Parallel Virtual Machine (PVM).

9.5.2 Ordering Problems

Many optimization problems fall into the category of ordering problems. The main characteristic of ordering problems is that the objective function depends only on the ordering of alleles. Thus, the fitness function F is equal to F(o), where F(o) gives the objective function value and the allele values are not independent of each other. As ordering problems do not lend themselves to binary string representation, a permutation representation seems the most natural way of coding the problems. But this type of representation is not consistent with the conventional GA crossover operator. This is illustrated by the following TSP example. Basically, TSP involves finding a route through different cities (i.e. an order in which the cities are visited) such that the total distance traveled by the salesman is minimized. Each chromosome then represents a particular tour, having a specific cost associated with it. Let the number of cities be 8 and the two chromosomes selected for crossover be denoted by tour 1 and tour 2.

tour1 1 4 5 6 2 3 | 8 7
tour2 5 6 3 2 7 8 | 4 1

After performing crossover, the new tours are:

Newtour 1 1 4 5 6 2 3 | 4 1
Newtour 2 5 6 3 2 7 8 | 8 7

The new tours generated after performing regular crossover operation are not valid tours because cities 1 and 4 are visited twice in the first offspring and cities 7 and 8 are visited twice in the second offspring. Further, in the new tour 1, cities 8 and 7 are not visited at all. Similarly, in new tour 2, cities 1 and 4 are not visited at all. In order to avoid this, modifications to basic crossover operators have been developed by several researchers. The commonly used crossover operators for permutation representation are PMX, OX, CX etc. Other operators have been developed which use a different (non-permutation) representation for the chromosomes.

Table 9.18 Different Crossover Operators and their Complexity

Crossover Operator	Reference	Complexity
Partially matched crossover (PMX)	[2]	$O(L^3)$
Order crossover (OX)	[12]	$O(L^3)$
Order crossover #2 (OX2)	[12]	$O(L^2)$
Cyclic crossover (CX)	[2]	$O(L^2)$
Enhanced Edge Recombination (EER)	[12]	$O(L^2)$
Position Transition Crossover (PTX)	[12]	$O(L^2)$
Intersection Crossover (IX)	[12]	$O(L^3)$
Union crossover (UX)	[12]	$O(L^3)$

(L is the length of a chromosome.)

The complexity of all the commonly used operators for ordering problems are summarized in Table 9.18.

It can be observed that for problems with large chromosomes, these operators will be computationally very expensive. This is due to the following reasons:

- Allele values in chromosomes are not independently modifiable.
- Many of these operators do not preserve the basic GA property of combining short defining length, low order and high performance schemata to generate new offsprings. Thus, it can be seen that there is no single crossover operator that combines simplicity and low computational complexity and moreover, is effective in obtaining an optimal solution to the problem.

9.5.3 Traveling Salesman Problem

The most well known problem in the class of ordering problems is TSP. Apart from the academic interest in TSP, there is also considerable interest in solving real-life ordering problems such as bin-packing and job shop scheduling. As mentioned before, TSP involves finding an optimal tour of cities such that the total distance traveled is minimized. The basic TSP can have several variations in terms of constraints (e.g. each city can be visited exactly once etc.) and parameters like city coordinates (whether the cities are arranged in a straight line, circle or on a 2-D plane). In this application, the following three variations of TSP are solved as follows:

Problem 1 : An N-city Hamiltonian Path TSP.

In this Hamiltonian Path TSP, the salesman can start at any city and finish at any other city visiting every other city exactly once. The N cities lie along the real line at the integers 1.. ..N. mere are two equivalent but opposite optimal routes $(a_1, a_2, a_{3,.....}a_N)$ and $(a_N, a_{N-1}, a_{N-2}, ------, a_1)$. Each of these two routes have a length of N-l units.

Problem 2 : An N-city Hamiltonian Cycle TSP.

In the Hamiltonian Cycle TSP, the salesman must start and finish at the same city. The N cities are spaced equally and lie on the edge of a circle. Journeys are measured around the circle. There are2*N equivalent optimal routes since the salesman can start at any city and travel in either the clockwise or anti-clockwise direction.

Problem 3: Oliver's N-city Hamiltonian Cycle TSP.

This problem is identical to the N-city Hamiltonian Cycle TSP except that the cities are located on a 2-D plane. Thus, each city has a pair of (x,y) coordinates associated with it.

Conventional Genetic Algorithm

In a conventional GA, the basic operations to be performed are :

Solution Representation (Coding)

Coding of the problem represents a solution for the problem. It can be represented as a string of binary numbers or integer numbers depending upon the type of problem being solved. This code is called chromosome.

Objective Function Evaluation

Each chromosome has an associated objective function value called fitness value. How the fitness value is obtained depends entirely on the chromosome representation for the problem. GA finds the fitness value of a chromosome based on the objective function specified for a given problem. A good chromosome is one which has high (low) fitness value depending on whether the problem is to maximize (minimize) some parameters.

Genetic Operations

A population of chromosomes at a given stage of GA is referred to as a generation. Since the population size in each generation is fixed, only a finite number of relatively good chromosomes can be copied in the mating pool depending on their fitness values. Chromosomes with higher (lower) fitness values contribute more copies to the mating pool. This is achieved by assigning a proportionately higher probability to a chromosome with higher fitness value.

Selection

Selection uses the fitness value of a chromosome to determine whether it should go to the mating pool or not. Roulette wheel selection mechanism is the commonly used method for selecting chromosomes. This method ensures that highly fit chromosomes (with relatively high (low) fitness value for maximization (minimization)) have a higher number of offsprings in the mating pool. Each chromosome (i) in the current generation is allotted a roulette wheel slot in proportion to its fitness value. This proportion P_i can be defined as follows:

P_1 = Fitness value of chromosome i/$\sum$ (fitness of all the chromosome in any generation) When the Roulette wheel is spun, Pi helps in getting better chromosomes in the mating pool.

Crossover

To perform crossover, initially, two chromosomes are selected randomly from the mating pool and then a crossover site C is selected at random in the interval [l, L-l], where L is the length of the chromosome. Two new chromosomes called offsprings are then obtained by swapping all the characters between position C and L. This can be shown using two chromosomes A and B, each of length 6. Assume C to be 3.

Chromosome A: 1 1 1 I 0 0 0
Chromosome B: 0 0 0 1 1 1 1

The two offsprings obtained from the two strings above are:

Chromosome C: 1 1 1 | 1 1 1
Chromosome D: 0 0 0 1 0 0 0

where the left part of the string in C and D is contributed by A and B respectively where as the right parts are contributed by B and A respectively.

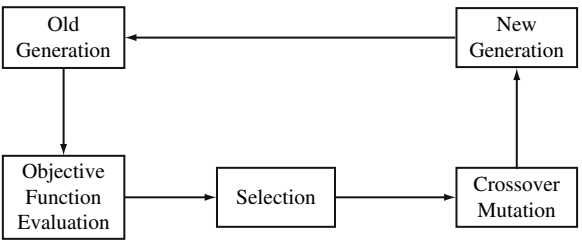


Fig. 9.19 Schematic of Traditional GA

Mutation

The combined operation of selection and crossover may sometimes lose some potentially good chromosomes. To overcome this problem, mutation is used. This is implemented by complementing a bit (0 to 1 and vice versa) to ensure that good chromosomes are not permanently lost.

A schematic of a traditional GA is shown in Figure 9.19.

9.5.4 Distributed Genetic Algorithm

Complex crossover techniques have to be used with traditional GA to solve ordering problems. Moreover, it suffers from two major drawbacks namely, critical dependence of its performance on probability of crossover P_c and probability of mutation P_m values and lack of consistency of the solution. Generally, considerable effort has to be made to fine tune the parameters (like P_c, P_m, etc.) in order to get a high quality solution.

To circumvent these drawbacks, a distributed implementation of GA which guarantees a consistent, high quality solution without requiring fine tuning of the parameters is developed. Further, the distributed GA can be used to solve ordering problems with a simple crossover technique. The distribution environment used in this scheme is the Parallel Virtual Machine (PVM). Various features provided by PVM have been exploited to incorporate user transparency and fault tolerance in this distributed scheme.

Distribution Scheme

In this scheme, a single GA process is not distributed across the network, rather several GA processes are executed in parallel. These processes periodically interact with each other to exchange information. The number of processes executed (denoted by P) is kept variable and has to be specified by the user before each run. PVM uses a hostfile to identify the machines to be used and allocates the various processes to the machines specified in the hostfile. Currently, the machines are allocated in the order in which they are listed in the hostfile and factors like machine load etc. are not taken into account during allocation. In order to avoid overloading a machine, each process is executed on a different machine. This requires that the number of machines used M should be at least equal to or greater than P, Thus, if M is greater than or equal to P, then each process gets assigned to a different machine. As the value of P can be varied for each run, it can be controlled such that it never exceeds M, which indicates the current availability of machines i.e. how many machines are currently up and are available for use.

Interprocess Interaction

This scheme aims to improve the performance of GA through the exchange of useful information between the various GA processes. For the exchange to be useful, it is important that each GA has different parameter values for P_c, and P_m, so that the search becomes more efficient. In order to make the behavior of the GA independent of P_c and P_m, values, each GA process uses a different set of values for P_c and P_m. This scheme also allows each GA process to have a different population size. However, the number of generations must be the same for each process. After each generation, every process multicasts a chromosome and its associated objective function value to all other processes. Thus, after each generation, every process receives (P-l) foreign chromosomes from other processes. The chromosome to be transmitted by a process is selected using the Roulette Wheel mechanism. The chromosome selected by the roulette wheel mechanism survives if it is truly a good chromosome else it gets eliminated in successive generations.

Transmission of a chromosome selected by Roulette Wheel is preferred over transmission of the chromosome which has highest (lowest) objective function value, because the chromosome that has the highest (lowest) objective function value may not necessarily be the best chromosome eventually. Also, transmission of the best chromosome may lead the GA processes to saturate prematurely because of the presence of local maxima (minima).

Figure 9.20 illustrates the GA processes executing on four different machines. As shown in Figure 9.20, each GA process uses a different set of P_c and P_m, values. The multicast facility provided in PVM is used by the sender GA process to transmit a chromosome to all the other processes. The details of establishing the distributed system (i.e. initialization) and Interprocess communication protocol are given below:

Initialization: The distributed system of several, concurrent GA processes is established by initiating a single GA process (termed as the parent GA process) manually. This GA process in turn tries to spawn the specified number of processes on different machines (if possible). The global parameter P is set to the actual number of processes that could be spawned. Thus, even if some of the machines are down, the set of working machines are used to establish the distributed system. In PVM, every process is assigned a unique identification number shown as a host id. The parent process multicasts an array of the task ids of the entire GA processes so that all the processes can exchange messages using the task ids. Once a process receives the host id array, it searches the array for its own host id. The index of the array element found becomes the value of a global parameter id. Thus, each process has a unique value of id in the order in which it was spawned. The parent GA process has id equal to 0. Once all the processes have been spawned and the value of id established for each process, all the processes become equal in status and can interact with each other as peers. A master-slave relationship is deliberately avoided to prevent a single machine from becoming the bottleneck in the system.

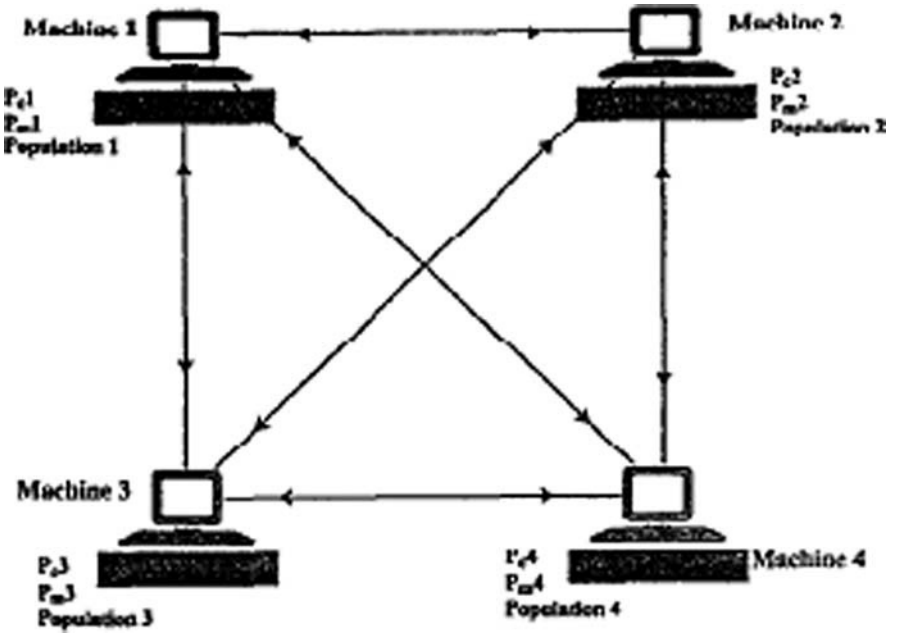


Fig. 9.20 Interaction of the GA processes in distributed implementation

Interprocess Communication Protocol

The communication protocol used can be explained in terms of an algorithm send-and-receive. The algorithm is so named because every GA process sends information to and also, receives information from all the other GA processes.

Algorithm send and receive:

begin send and receive
i = 1;
count = 0;
mark all the processes as active;
while (count < P) do
begin
if (count = id) then
begin
Multicast a chromosome selected by Roulette Wheel mechanism
Multicast the objective function value of the chromosome selected.
end
else
begin
check if all the GA processes are active
if there are any GA processes that have exited then mark them as inactive and update the status of the distributed system to contain only the inactive processes.

if $\text{sender}_{\text{count}}$ is not marked as inactive
begin
receive a chromosome from sender,,
receive its objective function value from $\text{sender}_{\text{count}}$,
end
count = count+1
end
end send-and-receive

Explanation of the Protocol: Initially, for all processes, count is initialized to zero. The main while loop is executed till count is less than P, In each iteration of the loop, count is tested for equality with id, and if they are equal, then the process is ready for transmission. Thus, the process with a value of 0 for id (which is the parent GA process) transmits a chromosome and its associated objective function value first. After the parent process has transmitted, it updates the count and waits to receive chromosomes from other processes. Every time a process receives a chromosome and its associated objective function value it increments the count. After incrementing the count, a check is made to test if the count is equal to id or not. If they are equal, then it is the process's turn to transmit. Since each process has its own copy of count which is updated only when information is received from a process, there is no possibility of deadlock, if all the processes remain active. But there is a possibility of deadlock, if, during execution, one or more processes exit abnormally, due to host failure or some other reason. For example, if a GA process with id $= 2$ is supposed to multicast information, but before doing so, is killed because of a host failure or some other reason. Then, all the other processes waiting to receive information from this GA process would wait forever and the system would be deadlocked. To avoid this possibility, a check is made for all inactive processes, before attempting a receive from any process.

PVM provides a facility that allows processes to be notified of certain events like a particular task exiting, a host getting deleted from the virtual machine etc. This facility is used in this scheme to detect failed processes so that the distributed system can be updated dynamically to avoid deadlock.

The flowchart in Figure 9.21 illustrates the distributed implementation of GA. As shown in the figure, an initialization step is required to establish the distributed system. Before adding new offsprings to the new generation, the send and receive protocol is invoked by each GA process to exchange chromosomes and their objective function values with all the other processes. When a chromosome is received, it replaces the worst chromosome (i.e. the chromosome having the minimum (maximum) objective function value) in the population. After each receive, the array of chromosomes is sorted in decreasing (increasing) order of their objective function values such that the worst chromosome is always at the end of the array. Replacement of the worst chromosome ensures that the best chromosomes are preserved in the population. The send and receive protocol is invoked in each generation. Thus, it is important that each GA process be invoked with the same number of generations. However, the population size of the various processes could be kept different.

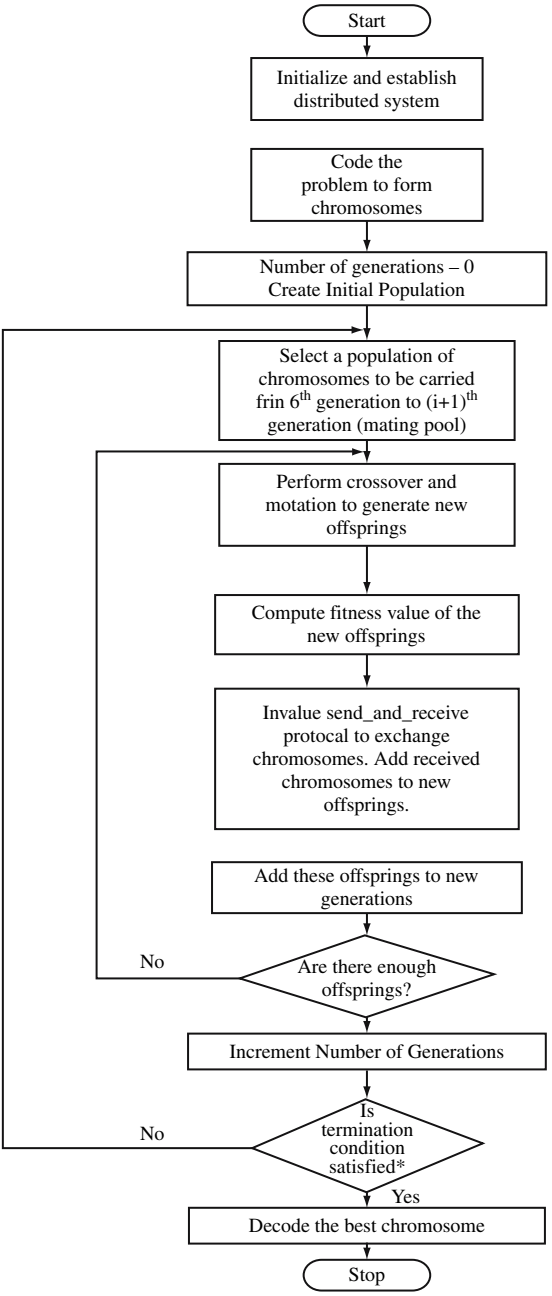

Start
Initialize and establish distributed system
Code the problem to form chromosomes
Number of generations – 0
Create Initial Population
Select a population of chromosomes to be carried frin 6th generation to (i+1)th generation (mating pool)
Perform crossover and motation to generate new offsprings
Compute fitness value of the new offsprings
Invalue send_and_receive protocal to exchange chromosomes. Add received chromosomes to new offsprings.
Add these offsprings to new generations
No
Are there enough offsprings?
Increment Number of Generations
No
Is termination condition satisfied*
Yes
Decode the best chromosome
Stop

Fig. 9.21 Distributed Genetic Algorithm

Crossover Scheme for Distributed GA

It can be observed that all the existing crossover operators for ordering problems have high computational complexity and are more complex to implement than the conventional crossover operator. Moreover, the computational complexity becomes especially significant for large size problems. In this application, a simple crossover scheme (termed as "swap crossover") is used to solve the TSP. In this scheme, integer representation is used for the chromosomal coding. Thus, for a 4-city TSP, if the chromosome is 4 3 1 2, then it implies that city 4 is visited first, city 3 next and so on. The swap crossover scheme is radically different from other crossover schemes in that it operates only on one chromosome. The operation of the crossover scheme is described as follows. Instead of selecting two chromosomes from the mating pool, only one chromosome is selected. Crossover "sites" within the selected chromosome are chosen at random. Then, the contents at these two sites are swapped. The following example illustrates this further. Consider that for a 4-city TSP, the selected chromosome Cl is:

Cl:	5	2	4	3
	site1	site2	site3	site4

Assume that sites 1 and 4 are chosen at random.
Then after crossover, the new chromosome generated C2 is:

C2:	3	2	4	5
	site1	site2	site3	site4

It is difficult to visualize swap crossover as a valid crossover technique because only one chromosome participates in the process. It can be argued that such an operation cannot generate a better chromosome with any certainty. Although this may be true in a conventional GA context, it is an effective technique when incorporated within the Distributed GA scheme. There are two reasons for this. The nature of coding in TSP and other ordering problems is such that a single chromosome possesses all the information that is needed to create a "perfect" child. The only requirement is to place the information in proper order to optimize the objective function. This self-contained nature of a chromosome in ordering problems is exploited in Distributed GA to obtain a high-quality solution. This is due to the fact that the efficiency with which the problem space is searched in a Distributed GA is greatly improved as compared to that of conventional GA. Also, the different values of P_c, and P_m, used by the concurrently executing GAs ensure diversity in chromosomes and reduce the probability of the GAs getting stuck in local minima (maxima). The mutation technique used in this scheme is slightly different from conventional mutation. Instead of randomly modifying an existing chromosome, a foreign chromosome is inserted into the population, replacing the worst chromosome (i.e. one having the worst fitness value). This foreign chromosome is generated randomly. The next section discusses the performance of both conventional GA and Distributed GA using the above crossover and mutation schemes.

9.5.4.1 Results For Various Problems

The three variations of TSP (Problems l–3), have been solved to demonstrate the usefulness and enhanced performance of Distributed GA. Results for the same problems using serial GA are also presented, and are compared with those obtained through distributed GA. The utility and improved performance of distributed GA is demonstrated in terms of the following qualities:

- High quality solution.
- Independence of GA solution from the values of P_c, and P_m.
- Consistency of the solution.

In order to demonstrate independence of the solution from the values of P_c, and P_m used, each GA process in the distributed implementation is executed with a different set of values for P, and P,. The values used are such that they cover a wide range of commonly used P_c, P_m, values. This is done to maintain diversity among the GA processes and to prevent biasing towards a particular set of P_c, P_m values.

In order to compare the performance of Serial GA with Distributed GA, several runs of both types of GA were carried out for different number of generations NG and population sizes PS. For a particular value of NG and PS, 20 runs for each problem instance of Problems l–3 were carried out. For Distributed GA, the number of machines used, M, in each run is 10. In each run of Distributed GA, each of the GA processes was executed with a different set of values for P_c and P_m. The value of P_c used for the various GA processes ranged from 0.5 to 0.9 and the value of P, ranged from 0.02 to 0.25.

Similarly, for each run of Serial GA, a different set of values for P, and P, were used. The range of these values was kept the same as that for Distributed GA. Each table presents the results obtained for Distributed GA and Serial GA for a particular problem instance. The MAX and MIN entries in each table give the maximum and minimum objective function values obtained over 20 runs. The AVG and STDEV entries in each table give the average and standard deviation of the objective function values obtained in 20 runs. The entries marked by * denote optimal values for the problem being solved.

Both the applications have been implemented on LAN of HP-UX workstations available at the University of Louisville. PVM has also been installed on the LAN for implementing the Distributed GA. Sections 7.1, 7.2, and 7.3 present a detailed discussion of the results obtained for Problems 1, 2 and 3, respectively.

Results for Hamiltonian Path TSP

Three instances of the Hamiltonian Path TSP have been solved using both Serial and Distributed GA. To demonstrate the utility of Distributed GA, the problem instances were chosen to be of varying sizes. Tables 9.19–9.21 present the results obtained for number of cities $N = 10$, 15, 20, 30 and 40 respectively. As can be observed from Tables 9.19–9.21, the average COST value found us-

Table 9.19 Results for 10 cities Hamiltonian path TSP

	DISTRIBUTED GA				SERIAL GA			
	MAX	MIN	AVG.	STDDEV.	MAX	MIN	AVG.	STDDEV.
NG=20 PS=10	12	9*	9.85	1.236	17	9*	12.98	2.7
NG=10 PS=20	12	9*	9.75	0.156	17	9*	11.76	2.489
NG=20 PS=40	10	9*	9.45	0.589	17	9*	11.53	2.227

ing Distributed GA is consistently better than the average COST value found using Serial GA. In order to compare consistency of the solutions found using Distributed GA and Serial GA, the standard deviation of the COST values over 20 runs is computed. The standard deviation of the objective function values over several runs of a GA is a measure of how consistently the GA finds the same solution or, in other words, how consistent the GA is. A low value of standard deviation implies high consistency. Also, since in each run, different values of P_c and P_m, are used, the standard deviation also reflects the dependence of the GA's performance on P_c and P_m. The effect of population size on the standard deviation is observed by computing the standard deviation for increasing values of PS, while maintaining NG constant. This can be observed from Tables 9.19–9.21 by looking at the STDEV entries. The entries show that in all the cases, the standard deviation for Distributed GA is significantly lower than the standard deviation found for Serial GA. This implies higher consistency of Distributed GA as compared to Serial GA and independence from the values of P_c, and P_m, used.

Table 9.20 Results for 20 cities Hamiltonian path TSP

	DISTRIBUTED GA				SERIAL GA			
	MAX	MIN	AVG.	STDDEV.	MAX	MIN	AVG.	STDDEV.
NG=10 PS=20	26	19*	22.75	3.064	37	19*	29.00	8.31
NG=20 PS=40	23	19*	21.19	1.461	37	19*	28.38	7.17
NG=40 PS=60	19*	19*	19	0	33	19*	24.91	6.615

Table 9.21 Results for 30 cities Hamiltonian path TSP

	DISTRIBUTED GA				SERIAL GA			
	MAX	MIN	AVG.	STDDEV.	MAX	MIN	AVG.	STDDEV.
NG=100 PS=40	49	39*	43.6	3.627	77	43	65	14.198
NG=100 PS=60	47	30*	42.0	3.322	55	59	50.1	6.159
NG=100 PS=80	43	35*	41.1	1.886	55	39	44.78	4.109

Results for Hamiltonian Cycle TSP

Three instances of the Hamiltonian Cycle TSP have been solved using both Serial and Distributed GA. As in the Hamiltonian Path TSP, the problem instances for Hamiltonian Cycle TSP were also chosen to be of varying sizes. Tables 9.22–9.24 present the results obtained for number of cities N = 10, 15, 20 and 40 respectively.

From Tables 9.22–9.24, it can be observed that the average CITY value of the solutions found using Distributed GA is consistently higher than the average CITY value found using Serial GA. The standard deviation of the CITY values was computed in the same manner as for Problem 1. The effect of population size on the standard deviation was observed by computing the standard deviation for increasing values of PS, while maintaining NG constant. It can be observed from Tables 9.22–9.24 that in all the cases, the standard deviation for Distributed GA is significantly lower than the standard deviation found for Serial GA. As for Problem 1, Distributed GA displays higher consistency as compared to Serial GA and independence from the values of P_c and P_m, used.

Table 9.22 Results for 10 cities Hamiltonian path TSP

	DISTRIBUTED GA				SERIAL GA			
	MAX	MIN	AVG.	STDDEV.	MAX	MIN	AVG.	STDDEV.
NG=10 PS=10	14	10	11.4	1.562	24	12	14.2	3.944
NG=10 PS=20	12	10	11	1.031	22	10	14.8	3.124
NG=10 PS=40	12	10	10.2	0.6	16	10	13	2.408

Table 9.23 Results for 10 cities Hamiltonian path TSP

	DISTRIBUTED GA				SERIAL GA			
	MAX	MIN	AVG.	STDDEV.	MAX	MIN	AVG.	STDDEV.
NG=20 PS=20	21	16	18.5	2.202	29	17	22.9	4.526
NG=20 PS=40	21	16	18.1	1.864	25	17	20.6	3.002
NG=20 PS=60	17	15	15.9	0.831	21	17	19.4	1.496

Table 9.24 Results for 10 cities Hamiltonian path TSP

	DISTRIBUTED GA				SERIAL GA			
	MAX	MIN	AVG.	STDDEV.	MAX	MIN	AVG.	STDDEV.
NG=20 PS=10	50	42	47.6	2.939	86	50	68.2	13.577
NG=20 PS=20	46	42	44.0	2.713	66	30	39.4	3.003
NG=20 PS=40	46	40	41.87	1.961	56	41	49.1	1.512

Results for Oliver's Hamiltonian Cycle TSP

Three instances of the Oliver 's Hamiltonian Cycle TSP have been solved using both Serial and Distributed GA. Table 9.25 gives the co-ordinates of cities used for N = 30.

As in the Hamiltonian Path TSP, the problem instances for Hamiltonian Cycle TSP were also chosen to be of varying sizes. Tables 9.26–9.28 present the results obtained for number of cities N = 10, 20 and 30 respectively. For N = 10, 20 the

Table 9.25 Coordinates for the cities for the 30 city Hamiltonian cycle TSP

City	Coord.	City	Coord.	City	Coord.
1	(87,7)	11	(51,69)	21	(4,50)
2	(91,38)	12	(54,62)	22	(13,40)
3	(83,46)	13	(51,67)	23	(14,40)
4	(71,44)	14	(37,64)	24	(24,42)
5	(64,60)	15	(43,94)	25	(25,38)
6	(68,58)	16	(2,99)	26	(41,26)
7	(83,69)	17	(7,64)	27	(41,26)
8	(81,76)	18	(22,60)	28	(44,50)
9	(74,78)	19	(21,62)	29	(58,33)
10	(71,71)	20	(18,54)	30	(62,52)

Table 9.26 Results for 10 city Oliver's Hamiltonian cycle TSP

	DISTRIBUTED GA				SERIAL GA			
	MAX	MIN	AVG.	STDDEV.	MAX	MIN	AVG.	STDDEV.
NG=20 PS=10	282.84	254.55	257.38	4.48	395.98	282.84	315.14	41.493
NG=20 PS=20	254.55	254.55	254.55	0	367.69	282.84	309.48	24.802
NG=20 PS=40	254.55	254.55	254.55	0	359.41	254.55	300.08	22.34

cities' coordinates lie along a straight line i.e. city 1 has coordinates (10, 10), city 2 has coordinates (20,20) and so on.

Table 9.27 Results for 20 city Oliver's Hamiltonian cycle TSP

	DISTRIBUTED GA				SERIAL GA			
	MAX	MIN	AVG.	STDDEV.	MAX	MIN	AVG.	STDDEV.
NG=60 PS=60	565.68	537.4	562.35	8.485	1187.94	791.36	1001.26	104.95
NG=60 PS=60	565.68	537.4	540.34	8.453	1074.8	735.39	970.75	101.94
NG=60 PS=60	537.4	537.4	537.4	0	989.54	650.53	782.19	99.21

The observations in this case are similar to the observations made for Problems 1 and 2. Distributed GA consistently yields a higher quality solution compared to Serial GA. The consistency of Distributed GA is also better than that of Serial GA. This can be observed by looking at the STDEV entries.

Table 9.28 Results for 30 city Oliver's Hamiltonian cycle TSP

	DISTRIBUTED GA				SERIAL GA			
	MAX	MIN	AVG.	STDDEV.	MAX	MIN	AVG.	STDDEV.
NG=150 PS=60	426.63	424	425.64	1.061	610.476	418.03	538.609	35.599
NG=150 PS=80	426.63	424	425.17	0.841	542.933	495.68	535.487	24.036
NG=150 PS=100	425.64	424	424.65	0.528	514.66	447.71	483.762	24.421

Conclusion

In this application, a scheme for implementing GA in a distributed environment is presented. Also, a new crossover scheme for ordering problems is presented. This crossover technique is shown to be effective in a Distributed GA environment. The advantage of using this crossover scheme is that it is very simple to implement and is computationally non-intensive. The main feature of our Distributed GA scheme is that it does not require specialized, expensive hardware and is simple to implement. Further, it is completely generic in nature i.e. it can be applied to a wide variety of problems, including ordering and non-ordering problems. From the results presented for TSP and Job Scheduling, it can be seen that Distributed GA yields a high quality and consistent solution. As Distributed GA is problem independent, there is a wide scope of its application. It can be used with the swap crossover scheme to solve a variety of ordering problems. Also, it can be applied to non-ordering problems for achieving a high quality and consistent solution without critical dependence upon the values of P_c, and P_m, used.

Appendix – A
Glossary

A

Adaptation – Adaptation is the process of adjustment to given (outer) conditions.
Allele – Allele, in biology, is the term given to the appropriate range of values for genes. In genetic algorithms, an allele is the value of the gene (or genes).
Allele Loss – Allele loss is the natural loss of traits in the gene pool over the generations of a run. Another term for allele loss is convergence. Severe allele loss results in a population incapable of solving the problem with the available gene pool.
Adaptive Behaviour – Underlying mechanisms that allow animals, and potentially, robots to adapt and survive in uncertain environments.
Artificial Intelligence – The study of how to make computers do things at the moment.
Artificial Life – The study of simple computer generated hypothetical life forms, i.e. life-as-it- could-be.
Automatically Defined Function (ADF) – Concept of modularization aiming at an efficiency increase in GP. ADFs are sub-trees which can be used as functions in main trees. ADFs are varied in the same manner as the main trees.

B

Building Block – A pattern of genes in a contiguous section of a chromosome which, if present, confers a high fitness to the individual. According to the building block hypothesis, a complete solution can be constructed by crossover joining together in a single individual with many building blocks which were originally spread throughout the population.
Baldwin Effect – If the ability to learn increases the fitness, survival, of an individual, then its offspring will have a high probability of having that ability to learn.

Boltzmann Selection – In Boltzmann selection, a method inspired by the technique of simulated annealing, selection pressure is slowly increased over evolutionary time to gradually focus the search. Given a fitness of f, Boltzmann selection assigns a new fitness, f_0, according to a differentiable function.
Birth Surplus – Surplus of offspring with respect to the parent population size, a necessary prerequisite for certain selection operators (comma selection).
Bit mutation/Bit flipping – Random negation of a bit position (gene) within the genome of an individual in the case of binary representation.
Bloat/Code bloat – Phenomenon of uncontrolled genome growth in GP individuals.
Block – Element of a tree-shaped GP individual. The block set consists of the terminal set and the function set.
Building Block Hypothesis (BBH) – It attempts to explain the functioning of a (binary) GA based on the schema theorem. The BBH is based on the assumption that beneficial properties of a parent are aggregated in (relatively) small code blocks at several locations within the genome. In the offspring they can be merged by crossover. Thus, the BBH suggests that the improved solution is assembled from "partial solutions," the so-called building blocks (see also genetic repair).

C

Canonical GA – Causality property of a system, that small variations in the cause only provoke small variations in the effect. In EAs variations of the cause correspond to changes of the genotype and variations of the effect to changes of the phenotype or of the respective objective function value (fitness). Although strong causality remarkably increases the efficiency of EAs, it is not always a prerequisite for their successful application.
Classifier System – A system which takes a set of inputs, produces a set of outputs which indicate some classification of the inputs. It is a special GA for evolving rule sets for rule based classifiers. An example might take inputs from sensors in a chemical plant, and classify them in terms of: 'running ok', 'needs more water', 'needs less water', 'emergency'.
Combinatorial Optimization – Some tasks involve combining a set of entities in a specific way (e.g. the task of building a house). A general combinatorial task involves deciding the specifications of those entities (e.g. what size, shape, material to make the bricks from), and the way in which those entities are brought together (e.g. the number of bricks, and their relative positions). If the resulting combination of entities can in some way be given a fitness score, then combinatorial optimization is the task of designing a set of entities, and deciding how they must be configured, so as to give maximum fitness.
Chromosome – Normally, in genetic algorithms the bit string which represents the individual. In genetic programming the individual and its representation are usually the same, both being the program parse tree. In nature many species store their genetic information on more than one chromosome.

Convergence – It is the tendency of members of the population to be the same. It may be used to mean either their representation or behavior. It indicates that a genetic algorithm solution has been reached.
Convergence Velocity – The rate of error reduction.
Cooperation – The behavior of two or more individuals acting to increase the gains of all participating individuals.
Crossover – A reproduction operator which forms a new chromosome by combining parts of each of the two 'parent' chromosomes. The simplest form is single-point CROSSOVER, in which an arbitrary point in the chromosome is picked. All the information from parent A is copied from the start up to the crossover point, then all the information from parent B is copied from the crossover point to the end of the chromosome. The new chromosome thus gets the head of one parent's chromosome combined with the tail of the other. Variations use more than one crossover point, or combine information from parents in other ways.
Cellular Automata – A regular array of identical finite state automata whose next state is determined solely by their current state and the state of their neighbours. The most widely seen is the game of life in which complex patterns emerge from a (supposedly infinite) square lattice of simple two state (living and dead) automata whose next state is determined solely by the current states of its four closes neighbours and itself.
Classifiers – An extension of genetic algorithms in which the population consists of a co-operating set of rules (i.e. a rulebase) which are to learn to solve a problem given a number of test cases. Between each generation the population as a whole is evaluated and a fitness is assigned to each rule using the bucket-brigade algorithm or other credit sharing scheme (e.g. the Pitt scheme). These schemes aims to reward or punish rules which contribute to a test case according to how good the total solution is by adjusting the individual rules fitness. At the end of the test data a new generation is created using a genetic algorithm as if each rule were independent using its own fitness (measures may be taken are taken to ensure a given rule only appears once in the new population).
Co-evolution – Two or more populations are evolved at the same time. Often the separate populations compete against each other.

D

Darwinism – Theory of evolution, proposed by Darwin, that evolution comes through random variation of heritable characteristics, coupled with natural selection (survival of the fittest). A physical mechanism for this, in terms of Genes and Chromosomes, was discovered many years later. Darwinism was combined with the selectionism of Weismann and the genetics of Mendel to form the Neo-Darwinian Synthesis during the 1930s-1950s by T. Dobzhansky, E. Mayr, G. Simpson, R. Fisher, S. Wright, and others.

Deme – A separately evolving subset of the whole population. The subsets may be evolved on different computers. Emigration between subset may be used.
Diploid – This refers to a cell which contains two copies of each chromosome. The copies are homologous i.e. they contain the same genes in the same sequence. In many sexually reproducing species, the genes in one of the sets of chromosomes will have been inherited from the father's gamete (sperm), while the genes in the other set of chromosomes are from the mother's gamete (ovum).
DNA – Deoxyribonucleic Acid, a double stranded macromolecule of helical structure (comparable to a spiral staircase). Both single strands are linear, unbranched nucleic acid molecules build up from alternating deoxyribose (sugar) and phosphate molecules. Each deoxyribose part is coupled to a nucleotide base, which is responsible for establishing the connection to the other strand of the DNA. The four nucleotide bases Adenine (A), Thymine (T), Cytosine (C) and Guanine (G) are the alphabet of the genetic information. The sequences of these bases in the DNA molecule determines the building plan of any organism.
Disruptive Selection – Individuals at both extremes of a range of phenotypes are favored over those in the middle. The evolutionary significance of disruptive selection lies in the possibility that the gene pool may become split into two distinct gene pools. This may be a way in which new species are formed.
Diversity Rate – Respective measure (s) of the genotypic difference of individuals.

E

Elitism – Elitism (or an elitist strategy) is a mechanism which is employed in some EAs which ensures that the chromosomes of the most highly fit member(s) of the population are passed on to the next generation without being altered by Genetic Operators. Using elitism ensures that the minimum fitness of the population can never reduce from one generation to the next. Elitism usually brings about a more rapid convergence of the population. In some applications elitism improves the chances of locating an optimal individual, while in others it reduces it.
Environment – Environment surrounds an organism. Can be 'physical' (abiotic), or biotic. In both, the organism occupies a niche which influences its fitness within the total environment. A biotic environment may present frequency-dependent fitness functions within a population, that is, the fitness of an organism's behavior may depend upon how many others are also doing it. Over several generations, biotic environments may foster co-evolution, in which fitness is determined with selection partly by other species.
Epistasis – A "masking" or "switching" effect among genes. A gene is said to be epistatic when its presence suppresses the effect of a gene at another locus. Epistatic genes are sometimes called inhibiting genes because of their effect on other genes which are described as hypostatic. Epistasis is referred to any kind of strong interaction among genes, not just masking effects. A possible definition is: "Epistasis is the

interaction between different genes in a chromosome. It is the extent to which the contribution to fitness of one gene depends on the values of other genes." Problems with little or no epistasis are trivial to solve (hillclimbing is sufficient). But highly epistatic problems are difficult to solve, even for GAs. High epistasis means that building blocks cannot form, and there will be deception.

Evolution – That process of change which is assured given a reproductive population in which there are varieties of individuals, with some varieties being heritable, of which some varieties differ in fitness (reproductive success).

Evolution Strategy (ES) – A type of evolutionary algorithm developed in the early 1960s in Germany. It employs real-coded parameters, and in its original form, it relied on mutation as the search operator, and a population size of one. Since then it has evolved to share many features with genetic algorithms.

Evolutionarily Stable Strategy – A strategy that performs well in a population dominated by the same strategy. Or, in other words, an 'ESS' is a strategy such that, if all the members of a population adopt it, no mutant strategy can invade.

Evolutionary Algorithm (EA) – A collective term for all variants of (probabilistic) optimization and approximation algorithms that are inspired by Darwinian evolution. Optimal states are approximated by successive improvements based on the variation-selection-paradigm. Thereby, the variation operators produce genetic diversity and the selection directs the evolutionary search.

Evolutionary Computation (EC) – Computation based on evolutionary algorithms. EC encompasses methods of simulating evolution on a computer. The term is relatively new and represents an effort bring together researchers who have been working in closely related fields but following different paradigms. The field is now seen as including research in genetic algorithms, evolution strategies, evolutionary programming, artificial life.

Evolutionary Programming (EP) – It is a stochastic optimization strategy, which is similar to Genetic Algorithms, but dispenses with both "genomic" representations and with crossover as a reproduction operator. It is a variant of EA, which, like ES, operates on the "natural" problem representation. Only mutation is used as the variation operator together with tournament selection; recombination is *not* employed.

Evolution Strategy – Evolution strategy is a variant of EA, which generally operates on the "natural" problem representation (no genotype-phenotype mapping for object parameters). An individual consists of a set of object parameters, the corresponding value of the objective function and a set of (endogenous) strategy parameters. The ES employs mutation and recombination as variation operators.

Evolutionary Systems – A process or system which employs the evolutionary dynamics of reproduction, mutation, competition and selection. The specific forms of these processes are irrelevant to a system being described as evolutionary.

Exploration – The process of visiting entirely new regions of a search space, to see if anything promising may be found there. Unlike exploitation, exploration involves leaps into the unknown. Problems which have many local maxima can sometimes only be solved by this sort of random search.

Evolvable Hardware – Evolvable hardware includes special devices, circuits or machines, which allow an implementation of the Darwinian evolution paradigm at the material level.

F

Fitness – A value assigned to an individual which reflects how well the individual solves the task in hand. A "fitness function" is used to map a chromosome to a fitness value. A "fitness landscape" is the hypersurface obtained by applying the fitness function to every point in the search space.
Function Optimization – For a function which takes a set of N input parameters, and returns a single output value, F, function optimization is the task of finding the set(s) of parameters which produce the maximum (or minimum) value of F. Function optimization is a type of value-based problem.
Function Set – The set of operators used in GP. These functions label the internal (non-leaf) points of the parse trees that represent the programs in the population. An example function set might be $\{+, -, ^*\}$.
Fitness Scaling – Fitness scaling converts the raw fitness scores that are returned by the fitness function to values in a range that is suitable for the selection function. The selection function uses the scaled fitness values to select the parents of the next generation. The selection function assigns a higher probability of selection to individuals with higher scaled values.
Function Set – The set of functions (atoms) available to the genetic program. The operators in the function set are used to make up the internal nodes of the parse tree.
Finite State Automation (FSA) or ***F***inite State Machine (FSM) – A machine which can be totally described by a finite set of states, it being in one of these states at any one time, plus a set of rules which determines when it moves from one state to another.
Fitness Function – A process which evaluates a member of a population and gives it a score or fitness. In most cases the goal is to find an individual with the maximum (or minimum) fitness.

G

Game Theory – A mathematical theory originally developed for human games, and generalized to human economics and military strategy, and to evolution in the theory of evolutionarily stable strategy. Game theory comes into its own wherever the optimum policy is not fixed, but depends upon the policy which is statistically most likely to be adopted by opponents.

Gamete – Cells which carry genetic information from their parents for the purpose of sexual reproduction. In animals, male gametes are called sperm, female gametes are called ova. Gametes have a haploid number of chromosomes.
Gene – A subsection of a chromosome which (usually) encodes the value of a single parameter.
Gene-Pool – The whole set of genes in a breeding population. The metaphor on which the term is based de-emphasizes the undeniable fact that genes actually go about in discrete bodies, and emphasizes the idea of genes flowing about the world like a liquid.
Generation – An iteration of the measurement of fitness and the creation of a new population by means of reproduction operators.
Generation Gap – Concept for describing overlapping generations (stationary EA). The generation gap is defined as the ratio of the number of offspring to the size of the parent population.
Genetic Algorithm (GA) – A type of evolutionary computation devised by John Holland. A model of machine learning that uses a genetic/evolutionary metaphor. Implementations typically use fixed-length character strings to represent their genetic information, together with a population of individuals which undergo crossover and mutation in order to find interesting regions of the search space.
Genetic Drift – Changes in gene/allele frequencies in a population over many generations, resulting from chance rather than selection, occurs most rapidly in small populations. It can lead to some Alleles becoming 'extinct', thus reducing the genetic variability in the population.
Genetic Programming (GP) – Genetic algorithms applied to programs. Genetic programming is more expressive than fixed-length character string GAs, though GAs are likely to be more efficient for some classes of problems.
Genetic Operator – A search operator acting on a coding structure that is analogous to a genotype of an organism (e.g. a chromosome).
Gene Pool – Total amount of all genes in a population.
Genotype – The genetic composition of an organism: the information contained in the genome.
Genome – The entire collection of genes (and hence chromosomes) possessed by an organism.
Global Optimization – The process by which a search is made for the extremum (or extrema) of a functional which, in evolutionary computation, corresponds to the fitness or error function that is used to assess the performance of any individual.
Generational GP – Generational genetic programming is the process of producing distinct generations in each iteration of the genetic algorithm.
Generation Equivalent – In a steady state GA, the time taken to create as many new individuals as there is in the population.
Genetic Program – A program produced by genetic programming.
Genetic Repair – Approach to explain the possible increase of performance by recombination. Accordingly, the task of recombination is to extract the genetic information common to the selected individuals, as this information is likely responsible for fitness increase. A perfect recombination operator should additionally reduce

those parts of the genome which are responsible for a decrease in fitness. This is, e.g., statistically realized by ES recombination operators in real-valued search spaces by (partially) averaging out the defective components.

H

Haploid – This refers to cell which contains a single chromosome or set of chromosomes, each consisting of a single sequence of genes. An example is a gamete. In EC, it is usual for individuals to be haploid. The solution to GA is a single set of chromosomes (one individual).
Hard Selection – Selection acts on competing individuals. When only the best available individuals are retained for generating future progeny, this is termed "hard selection." In contrast, "soft selection" offers a probabilistic mechanism for maintaining individuals to be parents of future progeny despite possessing relatively poorer objective values.
Hits – The number of hits an individual scores is the number of test cases for which it returns the correct answer (or close enough to it). This may or may not be a component of the fitness function. When an individual gains the maximum number of hits this may terminate the run.
Heuristics – These are suggestions that are written into the expert system program that suggests how to act when it doesn't really know what to do or doesn't have enough information.

I

Individual – A single member of a population. In EC, each individual contains a chromosome (or, more generally, a genome) which represents a possible solution to the task being tackled, i.e. a single point in the search space. Other information is usually also stored in each individual, e.g. its fitness.
Introns – Code sequences in the genome of GP individuals that do not influence the fitness of the individuals directly. Introns are regarded as one reason for code bloat.
Inversion – A reordering operator which works by selecting two cut points in a chromosome, and reversing the order of all the genes between those two points.

L

Learning Classifier System – A classifier system which "learns" how to classify its inputs. This often involves "showing" the system many examples of input patterns, and their corresponding correct outputs.

Local Search – Locating or approximating optimal states with variation operators or search strategies (not necessarily EAs), which explore only a limited part of the search space, the so-called (search) neighborhood. Thus, in general, local optima are found.

M

Mutation – A reproduction operator which forms a new chromosome by making (usually small) alterations to the values of genes in a copy of a single, parent chromosome.
Mutation Rate – Mutation probability of a single gene/object parameter of an individual. With respect to binary representation, the mutation rate is the probability of flipping a single bit position.
Mutation Strength – Usually the standard deviation of the normal distribution with which a single object parameter is mutated. Mutation strength is also a measure for the realized (search) neighborhood size.
Meta-GA – If a GA is used to set parameters or discover optimal settings for a second GA, the first one is known as a meta-GA.
Micro-GA – A micro-GA is a GA with a small population size (often 5) that has special reinitialization or mutation operators to increase diversity and prevent the natural convergence associated with small population sizes.
Migration – Migration is the exchange of individuals between subpopulations. Migration is used in the regional population model. The spread of information among subpopulations is influenced by the migration topology (i.e., which subpopulations exchange individuals), the migration interval (i.e., how often does an exchange take place), and the migration rate (i.e., number of individuals that are exchanged). These parameters determine whether the subpopulations evolve in a relatively independent way or rather behave like a panmictic population.
Multi-criteria Optimization – Optimization with regard to multiple objective functions aiming at a simultaneous improvement of the objectives. The goals are usually conflicting so that an optimal solution in the conventional sense does not exist. Instead one aims at e.g. Pareto optimality, i.e., one has to find the Pareto set from which the user can choose a qualified solution.
Multirecombination – Variation operator, which recombines more than two parents in order to produce one or more offspring.

N

Niche – In EC, it is often required to maintain diversity in the population. Sometimes a fitness function may be known to be multimodal, and it may be required to locate all the peaks. In this case consider each peak in the fitness function as analogous

to a niche. By applying techniques such as fitness sharing, the population can be prevented from converging on a single peak, and instead stable sub-Populations form at each peak. This is analogous to different species occupying different niches.
No Free Lunch Theorem – The NFL work of Wolpert and Macready is a framework that addresses the core aspects of search, focusing on the connection between fitness functions and effective search algorithms. The central importance of this connection is demonstrated by the No Free Lunch theorem which states that averaged over all problems, all search algorithms perform equally. This result implies that if a genetic algorithm is compared to some other algorithm (e.g., simulated annealing, or even random search) and the genetic algorithm performs better on some class of problems, then the other algorithm necessarily performs better on problems outside the class. Thus it is essential to incorporate knowledge of the problem into the search algorithm.

The NFL framework also does the following: it provides a geometric interpretation of what it means for an algorithm to be well matched to a problem; it provides information theoretic insight into the search procedure; it investigates time-varying fitness functions; it proves that independent of the fitness function, one cannot (without prior domain knowledge) successfully choose between two algorithms based on their previous behavior; it provides a number of formal measures of how well an algorithm performs; and it addresses the difficulty of optimization problems from a viewpoint outside of traditional computational complexity.
Normally Distributed – A random variable is normally distributed if its density function is described as f(x) = 1/sqrt(2*pi*sqr(sigma))* exp(−0.5*(x-mu)*(x-mu)/sqr(sigma)) where mu is the mean of the random variable x and sigma is the standard deviation.
Non-Terminal – Functions used to link parse tree together. This name may be used to avoid confusion with functions with no parameters which can only act as end points of the parse tree (i.e. leafs) and are part of the terminal set.

O

Object Variables – Parameters that are directly involved in assessing the relative worth of an individual.
Objective function/Quality function – Also known as goal function is the function to be optimized, depending on the object parameters (also referred to as search space parameters or phenotype parameters). The objective function constitutes the implementation of the problem to be solved. The input parameters are the object parameters. The output is the objective value representing the evaluation/quality of the individual/phenotype.
Offspring – An individual generated by any process of reproduction.
Optimization – The process of iteratively improving the solution to a problem with respect to a specified objective function.

Order-based Problem – A problem where the solution must be specified in terms of an arrangement (e.g. a linear ordering) of specific items, e.g. Traveling Salesman Problem, computer process scheduling. Order-based problems are a class of combinatorial optimization problems in which the entities to be combined are already determined.

P

Panmictic Population – Population in which there are no group structures or mating restrictions in the population (e.g. in terms of subpopulations), all individuals are potential recombination partners (global population model).
Parent – An individual who takes part in reproduction to generate one or more other individuals, known as offspring, or children.
Phenotype – The expressed traits of an individual.
Phylogenesis – Refers to a population of organisms. The life span of a population of organisms from pre-historic times until today.
Population – A group of individuals which may interact together, for example by mating, producing offspring, etc. Typical population sizes in EC range from one (for certain evolution strategies) to many thousands (for genetic programming).
Population size – Number of individuals in a population.
Parse Tree – A parse tree is the way the genetic programming paradigm represents the functions generated by a genetic program. A parse tree is similar to a binary decision tree and preorder traversal of the tree produces an S-expression that represents a potential solution in reverse polish notation.
Premature Convergence – A state when a genetic algorithm's population converges to something which is not the solution that is required.

R

Recombination – Recombination is also known as crossover.
Reordering – A reordering operator is a reproduction operator which changes the order of genes in a chromosome, with the hope of bringing related genes closer together, thereby facilitating the production of building blocks.
Reproduction – The creation of a new individual from two parents (sexual reproduction). Asexual reproduction is the creation of a new individual from a single parent.
Reproduction Operator – A mechanism which influences the way in which genetic information is passed on from parent(s) to offspring during reproduction. Operators fall into three broad categories: crossover, mutation and reordering operators.

S

Scaling function – Scaling function is the method for the transformation of objective values into fitness values (selection operator). Particularly when using fitness-proportionate selection it must be ensured that only positive fitness values are assigned to negative objective values; variants: rank-based selection, Boltzmann-selection.
Schema – A pattern of gene values in a chromosome, which may include 'dont care' states. Thus in a binary chromosome, each schema (plural schemata) can be specified by a string of the same length as the chromosome, with each character one of $\{0, 1, \#\}$. A particular chromosome is said to 'contain' a particular schema if it matches the schema (e.g. chromosome 01101 matches schema #1#0#). The 'order' of a schema is the number of non-dont-care positions specified, while the 'defining length' is the distance between the furthest two non-dont-care positions. Thus #1##0# is of order 2 and defining length 3.
Schema Theorem – Theorem devised by Holland to explain the behavior of GAs. In essence, it says that a GA gives exponentially increasing reproductive trials to above average schemata. Because each chromosome contains a great many schemata, the rate of schema processing in the population is very high, leading to a phenomenon known as implicit parallelism. This gives a GA with a population of size N a speedup by a factor of N cubed, compared to a random search.
Search Space – If the solution to a task can be represented by a set of N real-valued parameters, then the job of finding this solution can be thought of as a search in an N-dimensional space. This is referred to simply as the search space. More generally, if the solution to a task can be represented using a representation scheme, R, then the search space is the set of all possible configurations which may be represented in R.
Search Operators – Processes used to generate new individuals to be evaluated. Search operators in genetic algorithms are typically based on crossover and point mutation. Search operators in evolution strategies and evolutionary programming typically follow from the representation of a solution and often involve Gaussian or lognormal perturbations when applied to real-valued vectors.
Selection – The process by which some individual in a population are chosen for reproduction, typically on the basis of favoring individuals with higher fitness.
Simulation – The act of modeling a natural process.
Species – In EC the definition of "species" is less clear, since generally it is always possible for a pair of individuals to breed together. It is probably safest to use this term only in the context of algorithms which employ explicit speciation mechanisms.
Sub-Population – A population may be sub-divided into groups, known as sub-populations, where individuals may only mate with others in the same group. (This technique might be chosen for parallel processors). Such sub-divisions may markedly influence the evolutionary dynamics of a population. Sub-populations may be defined by various migration constraints: islands with limited arbitrary

migration; stepping-stones with migration to neighboring islands; isolation-by-distance in which each individual mate only with near neighbors.
Stochastic Universal Sampling – The individuals are mapped to contiguous segments of a line, such that each individual's segment is equal in size to its fitness exactly as in roulette-wheel selection. Here equally spaced pointers are placed over the line as many as there are individuals to be selected. Consider *NPointer* the number of individuals to be selected, then the distance between the pointers are *1/NPointer* and the position of the first pointer is given by a randomly generated number in the range $[0, 1/N_{\text{Pointer}}]$.
Simulated Annealing – Search technique where a single trial solution is modified at random. An *energy* is defined which represents how good the solution is. The goal is to find the best solution by minimizing the energy. Changes which lead to a lower energy are always accepted; an increase is probabilistically accepted. The probability is given by $\exp(-\Delta E/kT)$, where ΔE is the change in energy, k is a constant and T is the *Temperature*. Initially the temperature is high corresponding to a liquid or molten state where large changes are possible and it is progressively reduced using a *cooling schedule* so allowing smaller changes until the system *solidifies* at a low energy solution.

T

Terminal Set – The set of terminal (leaf) nodes in the parse trees representing the programs in the population. A terminal might be a variable, such as X, a constant value, or a function taking no arguments.
Traveling Salesman Problem – The traveling salesperson has the task of visiting a number of clients, located in different cities. The problem to solve is: in what order should the cities be visited in order to minimize the total distance traveled (including returning home)? This is a classical example of an order-based problem.
Terminals – Terminals are the numeric values, (variables, constants and zero argument functions) in the parse tree and are always external (leaf) nodes in the tree. The terminals act as arguments for the operator (atom) that is their parent in the tree.
Terminal Set – A set from which all end (leaf) nodes in the parse trees representing the programs must be drawn. A terminal might be a variable, a constant or a function with no arguments.
Termination Condition – The conditions which determine the termination of the evolutionary process (examples: number of objective function evaluations, maximum run time, and convergence in the fitness or search space).
Tournament Selection – A mechanism for choosing individuals from a population. A group (typically between 2 and 7 individuals) are selected at random from the population and the best (normally only one, but possibly more) is chosen.
Truncation Selection – Truncation selection is selection with a deterministic choice of the best μ individuals from the λ offspring (parents are not considered), necessary condition: $\lambda > \mu$.

V

Vector Optimization – It is typically, an optimization problem wherein multiple objectives must be satisfied. The goals are usually conflicting so that an optimal solution in the conventional sense does not exist. Instead one aims at e.g. Pareto optimality, i.e., one has to find the Pareto set from which the user can choose a qualified solution.

Appendix – B
Abbreviations

A list of abbreviations used in this book is given in this appendix.

2D-AR	–	**T**wo-**D**imensional **A**uto **R**egressive model
A/D	–	**A**ssembly/**D**isassembly process
ADF	–	**A**utomatically **D**efined **F**unctions
AGA	–	**A**daptive **G**enetic **A**lgorithms
AI	–	**A**rtificial-**I**ntelligence
ANN	–	**A**rtificial **N**eural **N**etwork**s**
ARO	–	**A**utomatic **R**e-used **O**utputs
ARR	–	**A**nnualised **R**ate of **R**eturn
BGP	–	**B**inary **G**enetic **P**rogramming
BRI	–	**B**est of **R**un **I**ndividual
CAD	–	**C**omputer **A**ided **D**esign
cGA	–	**c**ompact **G**enetic **A**lgorithm
CGP	–	**C**artesian **G**enetic **P**rogramming
CID	–	**C**ubic **I**nch **D**isplacement
CP	–	**C**artesian **P**rogram
CPGA	–	**C**oarse grained **P**arallel **G**enetic **A**lgorithm
CS	–	**C**lassifier **S**ystems
CSTR	–	**C**ontinuous **S**tirred **T**ank **R**eactor
CVaR	–	**C**onditional **Va**lue-at-**R**isk
DFX	–	**D**esign **f**or **X**
dGA	–	**d**istributed **G**enetic **A**lgorithm**s**
DJIA	–	**D**ow **J**ones **I**ndustrial **A**verage
DNA	–	**D**eoxyribo **N**ucleic **A**cid
dGA	–	**D**istributed **G**enetic **A**lgorithm
DTD	–	**D**ocument **T**ype **D**efinition
EA	–	**E**volutionary **A**lgorithm
EC	–	**E**volutionary **C**omputation
EDDIE	–	**E**volutionary **D**ynamic **D**ata **I**nvestment **E**valuator
EMH	–	**E**fficient **M**arket **H**ypothesis
EOS	–	**E**arth **O**bserving **S**atellites
EP	–	**E**volutionary **P**rogramming

ER – **E**dge **R**ecombination **C**rossover
ERC – **E**phemeral **R**andom **C**onstant
ES – **E**volution **S**trategies
ESFE – **E**quivalent **S**erial **F**unction **E**valuations
FEM – **F**inite **E**lements **M**ethod
FFT – **F**ast **F**ourier **T**ransform
FGA – **F**uzzy **G**enetic **A**lgorithms
FGP – **F**inancial **G**enetic **P**rogramming
FLC – **F**uzzy **L**ogic **C**ontrollers
GA – **G**enetic **A**lgorithms
GBD – **G**eneralized **B**enders **D**ecomposition
GDT – **G**enetic **D**ecision **T**ree
GFS – **G**enetic **F**uzzy **S**ystems
GGA – **G**enerational **G**enetic **A**lgorithm
GP – **G**enetic **P**rogramming
GPF – **G**enetic **P**rogramming **F**unctions
grael – **GRA**mmar **E**vo**L**ution
HBGA – **H**uman-**B**ased **G**enetic **A**lgorithm
HBSS – **H**euristic **B**iased **S**tochastic **S**earch
HC – **H**ill **C**limbing
HEP – **H**igh **E**nergy **P**hysics
HFBB – **H**ighly **F**it **B**uilding **B**lock
IAE – **I**ntegral of the **A**bsolute **E**rror
IAGA – **I**ntegrated **A**daptive **G**enetic **A**lgorithm
IMGA – **I**sland **M**odel **G**enetic **A**lgorithm
ITAE – **I**ntegral of the **T**ime weighted **A**bsolute **E**rror
JSSP – **J**ob **S**hop **S**cheduling **P**roblems
MGP – **M**eta-**G**enetic **P**rogramming
MINLP – **M**ixed Integer **N**onlinear **P**rogramming
mpGA – **M**assively **P**arallel **GA**
MOP – **M**ulti **O**bjective **P**roblems
MIMD – **M**ultiple **I**nstruction, **M**ultiple **D**ata **S**tream
MISD – **M**ultiple **I**nstructions, **S**ingle **D**ata **S**tream
MPI – **M**essage **P**assing **I**nterface
MSE – **M**ean **S**quare **E**rror
MVC – **M**odel, **V**iew and **C**ontroller
NLTF – **N**on-**L**inear **T**ransfer **F**unction
NN – **N**eural **N**etwork**s**
NLP – **N**onlinear **P**rogramming
NSGA – **N**on-dominated **S**orting based **G**enetic **A**lgorithm
OA – **O**uter **A**pproximation
OO – **O**bject **O**rientation
OX – **O**rder-based crossover
PADO – **P**arallel **A**lgorithm **D**iscovery and **O**rchestration
PCI – **P**robabilistically **C**omplete **I**nitialization

PDGP – **P**arallel **D**istributed **G**enetic **P**rogramming
PEA – **P**arallel **E**volutionary **A**lgorithms
PEI – **P**artially **E**numerative **I**nitialization
PGA – **P**arallel **G**enetic **A**lgorithm
PID – **P**roportional **I**ntegral **D**erivative
PVM – **P**arallel **V**irtual **M**achine
R4SDC – **R**adix-**4** **S**ingle-path **D**elay **C**ommutator
RNA – **R**ibo **N**ucleic **A**cid
ROI – **R**eturn **O**n **I**nvestment
RSI – **R**elative **S**trength **I**ndicators
RSCS – **R**educing **S**et of verti**C**e**S**
SA – **S**imulated **A**nnealing
SEIG – **S**elf-**E**xcited **I**nduction **G**enerator
SISD – **S**ingle **I**nstruction, **S**ingle **D**ata **S**tream
SIMD – **S**ingle **I**nstruction, **M**ultiple **D**ata **S**tream
SNR – **S**ignal to **N**oise **R**atio
SOM – **S**elf-**O**rganising **M**aps
SSGA – **S**teady **S**tate **G**enetic **A**lgorithm
SSR – **S**olid-**S**tate **R**ecorder
STAP – **S**pace-**T**ime **A**daptive **P**rocessing
STGP – **S**trongly **T**yped **G**enetic **P**rogramming
TSP – **T**ravelling **S**alesman **P**roblem
VaR – **V**alue **a**t **R**isk

Appendix – C
Research Projects

A brief description of research projects from various IEEE journals, Research centers and Universities are given in this chapter.

C.1 Evolutionary Simulation-based Validation

This project describes evolutionary simulation-based validation, a new point in the spectrum of design validation techniques, besides pseudo-random simulation, designer-generated patterns and formal verification. The proposed approach is based on coupling an evolutionary algorithm with a hardware simulator, and it is able to fit painlessly in an existing industrial flow. Prototypical tools were used to validate gate-level designs, comparing them against both their RT-level specifications and different gate-level implementations. Experimental results show that the proposed method is effectively able to deal with realistic designs, discovering potential problems, and, although approximate in nature, it is able to provide a high degree of confidence in the results and it exhibits a natural robustness even when used starting from incomplete information.

C.2 Automatic Generation of Validation Stimuli for Application-specific Processors

Microprocessor soft cores offer an effective solution to the problem of rapidly developing new system-on-a-chips. However, all the features they offer are rarely used in embedded applications, and thus designers are often involved in the challenging task of soft-core customization to obtain application-specific processors. This research project proposes a novel approach to help designers in the simulation-based validation of application-specific processors. Suitable input stimuli are automatically generated while reasoning only on the software application the processor is

intended to execute, while all the details concerning the processor hardware are neglected. Experimental results on a 8051 soft core show the effectiveness of the proposed approach.

C.3 Dynamic Prediction of Web Requests

As an increasing number of users access information on the World Wide Web, there is a opportunity to improve well known strategies for web pre-fetching, dynamic user modeling and dynamic site customization in order to obtain better subjective performance and satisfaction in web surfing. A new method is proposed to exploit user navigational path behavior to predict, in real-time, future requests. Real-time user adaptation avoids the use of statistical techniques on web logs by adopting a predictive user model. A new model is designed, derived from the Finite State Machine (FSM) formalism together with an evolutionary algorithm that evolves a population of FSMs for achieving a good prediction rate, and the performance of the prediction system is evaluated using the concepts of precision and applicability.

C.4 Analog Genetic Encoding for the Evolution of Circuits and Networks

This project describes a new kind of genetic representation called analog genetic encoding (AGE). The representation is aimed at the evolutionary synthesis and reverse engineering of circuits and networks such as analog electronic circuits, neural networks, and genetic regulatory networks. AGE permits the simultaneous evolution of the topology and sizing of the networks. The establishment of the links between the devices that form the network is based on an implicit definition of the interaction between different parts of the genome. This reduces the amount of information that must be carried by the genome, relatively to a direct encoding of the links. The application of AGE is illustrated with examples of analog electronic circuit and neural network synthesis. The performance of the representation and the quality of the results obtained with AGE are compared with those produced by genetic programming.

C.5 An Evolutionary Algorithm for Global Optimization Based on Level-set Evolution and Latin Squares

In this project, the level-set evolution is exploited in the design of a novel evolutionary algorithm (EA) for global optimization. An application of Latin squares leads to a new and effective crossover operator. This crossover operator can generate a

set of uniformly scattered offspring around their parents, has the ability to search locally, and can explore the search space efficiently. To compute a globally optimal solution, the level set of the objective function is successively evolved by crossover and mutation operators so that it gradually approaches the globally optimal solution set. As a result, the level set can be efficiently improved. Based on these skills, a new EA is developed to solve a global optimization problem by successively evolving the level set of the objective function such that it becomes smaller and smaller until all of its points are optimal solutions. Furthermore, it can be proved that the proposed algorithm converges to a global optimizer with probability one. Numerical simulations are conducted for 20 standard test functions. The performance of the proposed algorithm is compared with that of eight EAs that have been published recently and the Monte Carlo implementation of the mean-value-level-set method. The results indicate that the proposed algorithm is effective and efficient.

C.6 Imperfect Evolutionary Systems

In this project, a change from a perfect paradigm to an imperfect paradigm in evolving intelligent systems is proposed. An imperfect evolutionary system (IES) is introduced as a new approach in an attempt to solve the problem of an intelligent system adapting to new challenges from its imperfect environment, with an emphasis on the incompleteness and continuity of intelligence. An IES is defined as a system where intelligent individuals optimize their own utility, with the available resources, while adapting themselves to the new challenges from an evolving and imperfect environment. An individual and social learning paradigm (ISP) is presented as a general framework for developing IESs. A practical implementation of the ISP framework, an imperfect evolutionary market, is described. Through experimentation, the absorption of new information from an imperfect environment by artificial stock traders and the dissemination of new knowledge within an imperfect evolutionary market is demonstrated. Parameter sensitivity of the ISP framework is also studied by employing different levels of individual and social learning.

C.7 A Runtime Analysis of Evolutionary Algorithms for Constrained Optimization Problems

Although there are many evolutionary algorithms (EAs) for solving constrained optimization problems, there are few rigorous theoretical analyses. This project presents a time complexity analysis of EAs for solving constrained optimization. It is shown when the penalty coefficient is chosen properly, direct comparison between pairs of solutions using penalty fitness function is equivalent to that using the criteria “superiority of feasible point” or “superiority of objective function value.” This project analyzes the role of penalty coefficients in EAs in terms of time complexity.

The results show that in some examples, EAs benefit greatly from higher penalty coefficients, while in other examples, EAs benefit from lower penalty coefficients. This project also investigates the runtime of EAs for solving the 0–1 knapsack problem and the results indicate that the mean first hitting times ranges from a polynomial-time to an exponential time when different penalty coefficients are used.

C.8 Classification with Ant Colony Optimization

Ant colony optimization (ACO) can be applied to the data mining field to extract rule-based classifiers. The aim of this project is twofold. On the one hand, an overview of previous ant-based approaches to the classification task is provided and compared with state-of-the-art classification techniques, such as C4.5, RIPPER, and support vector machines in a benchmark study. On the other hand, a new ant-based classification technique is proposed, named AntMiner+. The key differences between the proposed AntMiner+ and previous AntMiner versions are the usage of the better performing ${cal M}{cal A}{cal X}$-${cal M}{cal I}{cal N}$ ant system, a clearly defined and augmented environment for the ants to walk through, with the inclusion of the class variable to handle multiclass problems, and the ability to include interval rules in the rule list. Furthermore, the commonly encountered problem in ACO of setting system parameters is dealt with in an automated, dynamic manner. The benchmarking experiments show an AntMiner+ accuracy that is superior to that obtained by the other AntMiner versions, and competitive or better than the results achieved by the compared classification techniques.

C.9 Multiple Choices and Reputation in Multiagent Interactions

Co-evolutionary learning provides a framework for modeling more realistic iterated prisoner's dilemma (IPD) interactions and to study conditions of how and why certain behaviors (e.g., cooperation) in a complex environment can be learned through an adaptation process guided by strategic interactions. The co-evolutionary learning of cooperative behaviors can be attributed to the mechanism of direct reciprocity (e.g., repeated encounters). However, for the more complex IPD game with more choices, it is unknown precisely why the mechanism of direct reciprocity is less effective in promoting the learning of cooperative behaviors. Here, the study suggests that the evolution of defection may be a result of strategies effectively having more opportunities to exploit others when there are more choices. It is found that these strategies are less able to resolve the intention of an intermediate choice, e.g., whether it is a signal to engender further cooperation or a subtle exploitation. A likely consequence is strategies adapting to lower cooperation plays that offer higher payoffs in the short-term view when they cannot resolve the intention of opponents. However, cooperation in complex human interactions may also involve

indirect interactions rather than direct interactions only. Following this, the co-evolutionary learning of IPD with more choices and reputation is studied. Here, current behavioral interactions depend not only on choices made in previous moves (direct interactions), but also choices made in past interactions that are reflected by their reputation scores (indirect interactions). The co-evolutionary learning of co-operative behaviors is possible in the IPD with more choices when strategies use reputation as a mechanism to estimate behaviors of future partners and to elicit mutual cooperation play right from the start of interactions. In addition, the impact of the accuracy of reputation estimation in reflecting strategy behaviors of different implementations and its importance for the evolution of cooperation is also studied. Finally it is proved that the accuracy is related to how memory of games from previous generations is incorporated to calculate reputation scores and how frequently reputation scores are updated.

C.10 Coarse-grained Dynamics for Generalized Recombination

An exact microscopic model for the dynamics of a genetic algorithm with generalized recombination is presented. Generalized recombination is a new model for the exchange of genetic material from parents to offspring that generalizes and subsumes standard operators, such as homologous crossover, inversion and duplication, and in which a particular gene in the offspring may originate from any parental gene. It is shown that the dynamics naturally coarse grains, the appropriate effective degrees of freedom being schemata that act as building blocks. It is shown that the Schema dynamics has the same functional form as that of strings and a corresponding exact Schema theorem is derived. To exhibit the qualitatively new phenomena that can occur in the presence of generalized recombination, and to understand the biases of the operator, a complete, exact solution for a two-locus model without selection is derived, showing how the dynamical behavior is radically different to that of homologous crossover. Inversion is shown to potentially introduce oscillations in the dynamics, while gene duplication leads to an asymmetry between homogeneous and heterogeneous strings. All non-homologous operators lead to allele "diffusion" along the chromosome. This project discusses how inferences from the two-locus results extend to the case of a recombinative genetic algorithm with selection and more than two loci providing evidence from an integration of the exact dynamical equations for more than two loci.

C.11 An Evolutionary Algorithm-based Approach to Automated Design of Analog and RF Circuits Using Adaptive Normalized Cost Functions

Typical analog and radio frequency (RF) circuit sizing optimization problems are computationally hard and require the handling of several conflicting cost criteria.

Many researchers have used sequential stochastic refinement methods to solve them, where the different cost criteria can either be combined into a single-objective function to find a unique solution, or they can be handled by multi-objective optimization methods to produce tradeoff solutions on the Pareto front. This project presents a method for solving the problem by the former approach. A systematic method for incorporating the tradeoff wisdom inspired by the circuit domain knowledge in the formulation of the composite cost function is proposed. Key issues have been identified and the problem has been divided into two parts: a) normalization of objective functions and b) assignment of weights to objectives in the cost function. A nonlinear, parameterized normalization strategy has been proposed and has been shown to be better than traditional linear normalization functions. Further, the designers' problem specific knowledge is assembled in the form of a partially ordered set, which is used to construct a hierarchical cost graph for the problem. The scalar cost function is calculated based on this graph. Adaptive mechanisms have been introduced to dynamically change the structure of the graph to improve the chances of reaching the near-optimal solution. A correlated double sampling offset-compensated switched capacitor analog integrator circuit and an RF low-noise amplifier in an industry-standard 0.18mum CMOS technology have been chosen for experimental study. Optimization results have been shown for both the traditional and the proposed methods. The results show significant improvement in both the chosen design problems.

C.12 An Investigation on Noisy Environments in Evolutionary Multi-objective Optimization

In addition to satisfying several competing objectives, many real-world applications are also characterized by a certain degree of noise, manifesting itself in the form of signal distortion or uncertain information. In this research project, extensive studies are carried out to examine the impact of noisy environments in evolutionary multi-objective optimization. Three noise-handling features are then proposed based upon the analysis of empirical results, including an experiential learning directed perturbation operator that adapts the magnitude and direction of variation according to past experiences for fast convergence, a gene adaptation selection strategy that helps the evolutionary search in escaping from local optima or premature convergence, and a possibilistic archiving model based on the concept of possibility and necessity measures to deal with problem of uncertainties. In addition, the performances of various multi-objective evolutionary algorithms in noisy environments, as well as the robustness and effectiveness of the proposed features are examined based upon five benchmark problems characterized by different difficulties in local optimality, non-uniformity, discontinuity, and non-convexity.

C.13 Interactive Evolutionary Computation-based Hearing Aid Fitting

An interactive evolutionary computation (EC) fitting method is proposed that applies interactive EC to hearing aid fitting and the method is evaluated using a hearing aid simulator with human subjects. The advantages of the method are that it can optimize a hearing aid based on how a user hears and that it realizes whatever + whenever + wherever (W3) fitting. Conventional fitting methods are based on the user's partially measured auditory characteristics, the fitting engineer's experience, and the user's linguistic explanation of his or her hearing. These conventional methods, therefore, suffer from the fundamental problem that no one can experience another person's hearing. However, as interactive EC fitting uses EC to optimize a hearing aid based on the user's evaluation of his or her hearing, this problem is addressed. Moreover, whereas conventional fitting methods must use pure tones and bandpass noise for measuring hearing characteristics, the proposed method has no such restrictions. Evaluating the proposed method using speech sources, it is found that better results are obtained than either the conventional method or the unprocessed case in terms of both speech intelligibility and speech quality. The method is also evaluated using musical sources, unusable for evaluation by conventional methods. The method also demonstrates that its sound quality is preferable to the unprocessed case.

C.14 Evolutionary Development of Hierarchical Learning Structures

Hierarchical reinforcement learning (RL) algorithms can learn a policy faster than standard RL algorithms. However, the applicability of hierarchical RL algorithms is limited by the fact that the task decomposition has to be performed in advance by the human designer. A Lamarckian evolutionary approach is proposed for automatic development of the learning structure in hierarchical RL. The proposed method combines the MAXQ hierarchical RL method and genetic programming (GP). In the MAXQ framework, a subtask can optimize the policy independently of its parent task's policy, which makes it possible to reuse learned policies of the subtasks. In the proposed method, the MAXQ method learns the policy based on the task hierarchies obtained by GP, while the GP explores the appropriate hierarchies using the result of the MAXQ method. To show the validity of the proposed method, simulation experiments were performed for a foraging task in three different environmental settings. The results show strong interconnection between the obtained learning structures and the given task environments. The main conclusion of the experiments is that the GP can find a minimal strategy, i.e., a hierarchy that minimizes the number of primitive subtasks that can be executed for each type of situation. The experimental results for the most challenging environment also show that the

policies of the subtasks can continue to improve, even after the structure of the hierarchy has been evolutionary stabilized, as an effect of Lamarckian mechanisms.

C.15 Knowledge Interaction with Genetic Programming in Mechatronic Systems Design Using Bond Graphs

This project describes a unified network synthesis approach for the conceptual stage of mechatronic systems design using bond graphs. It facilitates knowledge interaction with evolutionary computation significantly by encoding the structure of a bond graph in a genetic programming tree representation. On the one hand, since bond graphs provide a succinct set of basic design primitives for mechatronic systems modeling, it is possible to extract useful modular design knowledge discovered during the evolutionary process for design creativity and reusability. On the other hand, design knowledge gained from experience can be incorporated into the evolutionary process to improve the topologically open-ended search capability of genetic programming for enhanced search efficiency and design feasibility. This integrated knowledge-based design approach is demonstrated in a quarter-car suspension control system synthesis and a MEMS bandpass filter design application.

C.16 A Distributed Evolutionary Classifier for Knowledge Discovery in Data Mining

This project presents a distributed co-evolutionary classifier (DCC) for extracting comprehensible rules in data mining. It allows different species to be evolved cooperatively and simultaneously, while the computational workload is shared among multiple computers over the Internet. Through the intercommunications among different species of rules and rule sets in a distributed manner, the concurrent processing and computational speed of the co-evolutionary classifiers are enhanced. The advantage and performance of the proposed DCC are validated upon various datasets obtained from the UCI machine learning repository. It is shown that the predicting accuracy of DCC is robust and the computation time is reduced as the number of remote engines increases. Comparison results illustrate that the DCC produces good classification rules for the datasets, which are competitive as compared to existing classifiers in literature.

C.17 Evolutionary Feature Synthesis for Object Recognition

Features represent the characteristics of objects and selecting or synthesizing effective composite features are the key to the performance of object recognition. In this

project, a co-evolutionary genetic programming (CGP) approach to learn composite features for object recognition. The knowledge about the problem domain is incorporated in primitive features that are used in the synthesis of composite features by CGP using domain-independent primitive operators. The motivation for using CGP is to overcome the limitations of human experts who consider only a small number of conventional combinations of primitive features during synthesis. CGP, on the other hand, can try a very large number of unconventional combinations and these unconventional combinations yield exceptionally good results in some cases. Experimental results with real synthetic aperture radar (SAR) images show that CGP can discover good composite features to distinguish objects from clutter and to distinguish among objects belonging to several classes.

C.18 Accelerating Evolutionary Algorithms with Gaussian Process Fitness Function Models

This project presents an overview of evolutionary algorithms that use empirical models of the fitness function to accelerate convergence, distinguishing between evolution control and the surrogate approach. The Gaussian process model is described and proposed using it as an inexpensive fitness function surrogate. Implementation issues such as efficient and numerically stable computation, exploration versus exploitation, local modeling, multiple objectives and constraints, and failed evaluations are addressed. The resulting Gaussian process optimization procedure clearly outperforms other evolutionary strategies on standard test functions as well as on a real-world problem: the optimization of stationary gas turbine compressor profiles.

C.19 A Constraint-based Genetic Algorithm Approach for Mining Classification Rules

Data mining is an information extraction process that aims to discover valuable knowledge in databases. Existing genetic algorithms (GAs) designed for rule induction evaluates the rules as a whole via a fitness function. Major drawbacks of GAs for rule induction include computation inefficiency, accuracy and rule expressiveness. In this project, a constraint-based genetic algorithm (CBGA) approach is proposed to reveal more accurate and significant classification rules. This approach allows constraints to be specified as relationships among attributes according to predefined requirements, user's preferences, or partial knowledge in the form of a constraint network. The constraint-based reasoning is employed to produce valid chromosomes using constraint propagation to ensure the genes to comply with the

predefined constraint network. The proposed approach is compared with a regular GA and C4.5 using two UCI repository data sets. Better classification accurate rates from CBGA are demonstrated.

C.20 An Evolutionary Algorithm for Solving Nonlinear Bilevel Programming Based on a New Constraint-handling Scheme

In this project, a special nonlinear bilevel programming problem (nonlinear BLPP) is transformed into an equivalent single objective nonlinear programming problem. To solve the equivalent problem effectively, a specific optimization problem with two objectives is constructed. By solving the specific problem, the leader's objective value can be decreased and the quality of any feasible solution from infeasible solutions can be identified and the quality of two feasible solutions for the equivalent single objective optimization problem, force the infeasible solutions moving toward the feasible region, and improve the feasible solutions gradually. A new constraint-handling scheme and a specific-design crossover operator is proposed. The new constraint-handling scheme can make the individuals satisfy all linear constraints exactly and the nonlinear constraints approximately. The crossover operator can generate high quality potential offspring. Based on the constraint-handling scheme and the crossover operator, a new evolutionary algorithm is proposed and its global convergence is proved. A distinguishing feature of the algorithm is that it can be used to handle nonlinear BLPPs with nondifferentiable leader's objective functions. Finally, simulations on 31 benchmark problems, 12 of which have nondifferentiable leader's objective functions, are made and the results demonstrate the effectiveness of the proposed algorithm.

C.21 Evolutionary Fuzzy Neural Networks for Hybrid Financial Prediction

In this project, an evolutionary fuzzy neural network using fuzzy logic, neural networks (NNs), and genetic algorithms (GAs) is proposed for financial prediction with hybrid input data sets from different financial domains. A new hybrid iterative evolutionary learning algorithm initializes all parameters and weights in the five-layer fuzzy NN, then uses GA to optimize these parameters, and finally applies the gradient descent learning algorithm to continue the optimization of the parameters. Importantly, GA and the gradient descent learning algorithm are used alternatively in an iterative manner to adjust the parameters until the error is less than the required value. Unlike traditional methods, not only the data of the prediction factor are considered, but also the hybrid factors related to the prediction factor are considered.

Bank prime loan rate, federal funds rate and discount rate are used as hybrid factors to predict future financial values. The simulation results indicate that hybrid iterative evolutionary learning combining both GA and the gradient descent learning algorithm is more powerful than the previous separate sequential training algorithm described in.

C.22 Genetic Recurrent Fuzzy System by Coevolutionary Computation with Divide-and-Conquer Technique

A genetic recurrent fuzzy system which automates the design of recurrent fuzzy networks by a coevolutionary genetic algorithm with divide-and-conquer technique (CGA-DC) is proposed in this project. To solve temporal problems, the recurrent fuzzy network constructed from a series of recurrent fuzzy if-then rules is adopted. In the CGA-DC, based on the structure of a recurrent fuzzy network, the design problem is divided into the design of individual subrules, including spatial and temporal, and that of the whole network. Then, three populations are created, among which two are created for spatial and temporal subrules searches, and the other for the whole network search. Evolution of the three populations are performed independently and concurrently to achieve a good design performance. To demonstrate the performance of CGA-DC, temporal problems on dynamic plant control and chaotic system processing are simulated. In this way, the efficacy and efficiency of CGA-DC can be evaluated as compared with other genetic-algorithm-based design approaches.

C.23 Knowledge-based Fast Evaluation for Evolutionary Learning

The increasing amount of information available is encouraging the search for efficient techniques to improve the data mining methods, especially those which consume great computational resources, such as evolutionary computation. Efficacy and efficiency are two critical aspects for knowledge-based techniques. The incorporation of knowledge into evolutionary algorithms (EAs) should provide either better solutions (efficacy) or the equivalent solutions in shorter time (efficiency), regarding the same evolutionary algorithm without incorporating such knowledge. In this project, some of the incorporation of knowledge techniques for evolutionary algorithms are categorized and summarized and a novel data structure, called efficient evaluation structure (EES) is presented, which helps the evolutionary algorithm to provide decision rules using less computational resources. The EES-based EA is tested and compared to another EA system and the experimental results show the quality of this approach, reducing the computational cost about 50%, maintaining the global accuracy of the final set of decision rules.

C.24 A Comparative Study of Three Evolutionary Algorithms Incorporating Different Amounts of Domain Knowledge for Node Covering Problem

This project compares three different evolutionary algorithms for solving the node covering problem: EA-I relies on the definition of the problem only without using any domain knowledge, while EA-II and EA-III employ extra heuristic knowledge. In theory, it is proven that all three algorithms can find an optimal solution in finite generations and find a feasible solution efficiently; but none of them can find the optimal solution efficiently for all instances of the problem. Through experiments, it is observed that all three algorithms can find a feasible solution efficiently, and the algorithms with extra heuristic knowledge can find better approximation solutions, but none of them can find the optimal solution to the first instance efficiently. This project shows that heuristic knowledge is helpful for evolutionary algorithms to find good approximation solutions, but it contributes little to search for the optimal solution in some instances.

Appendix – D
MATLAB Toolboxes

A few MATLAB Toolboxes that are commonly used such as Genetic Algorithm and Direct Search Toolbox, Genetic and Evolutionary Algorithm Toolbox, Genetic Algorithm Toolbox and Genetic Programming toolbox are discussed in this appendix.

D.1 Genetic Algorithm and Direct Search Toolbox

Genetic Algorithm and Direct Search Toolbox is a collection of functions that extend the capabilities of Optimization Toolbox and the MATLAB numeric computing environment. Genetic Algorithm and Direct Search Toolbox includes routines for solving optimization problems using

- Direct search
- Genetic algorithm
- Simulated annealing

These algorithms enable the user to solve a variety of optimization problems that lie outside the scope of Optimization Toolbox. All the toolbox functions are MATLAB M-files made up of MATLAB statements that implement specialized optimization algorithms. The MATLAB code for these functions can be viewed using the statement

```
type function_name
```

The user can extend the capabilities of Genetic Algorithm and Direct Search Toolbox by writing user's own M-files, or by using the toolbox in combination with other toolboxes, or with MATLAB or Simulink.

The Genetic Algorithm and Direct Search Toolbox extends the optimization capabilities in MATLAB and the Optimization Toolbox with tools for using the genetic and direct search algorithms. These algorithms can be used for problems that are difficult to solve with traditional optimization techniques, including problems that are not well defined or are difficult to model mathematically. The user can also

use them when computation of the objective function is discontinuous, highly nonlinear, stochastic, or has unreliable or undefined derivatives.

The Genetic Algorithm and Direct Search Toolbox complements other optimization methods to help the user find good starting points. Then the traditional optimization techniques can be used to refine the solution. Toolbox functions, which can be accessed through a graphical user interface (GUI) or the MATLAB command line, are written in the open MATLAB language. This means that the user can inspect the algorithms, modify the source code, and create own custom functions.

Key Features include:

- Graphical user interfaces and command-line functions for quickly setting up problems, setting algorithm options, and monitoring progress
- Genetic algorithm tools with options for creating initial population, fitness scaling, parent selection, crossover, and mutation
- Direct search tools that implement a pattern search method, with options for defining mesh size, polling technique, and search method
- Ability to solve optimization problems with nonlinear, linear, and bound constraints
- Functions for integrating Optimization Toolbox and MATLAB routines with the genetic or direct search algorithm
- Support for automatic M-code generation

D.2 Genetic and Evolutionary Algorithm Toolbox

The GEATbx (Genetic and Evolutionary Algorithm Toolbox for use with MATLAB) contains a broad range of tools for solving real-world optimization problems. They not only cover pure optimization, but also the preparation of the problem to be solved, the visualization of the optimization process, the reporting and saving of results, and as well as some other special tools.

When using the GEATbx the user needs to know how to implement the problem ('Writing Objective Functions'). Another important aspect which must be considered is the format of the 'Variable Representation'.

Features of the GEATbx include

- real, integer, binary (linear and logarithmic scaling, gray coding) and permutation variable representation
- fitness assignment: linear/non-linear ranking
- multi-objective ranking: PARETO ranking, goal attainment, sharing
- selection: stochastic universal sampling, local, truncation, tournament selection
- recombination: discrete, intermediate, line, extended line, permutation/scheduling
- crossover: single/double point, shuffle, reduced surrogate

- mutation: binary, integer, real valued, permutation/scheduling
- reinsertion: global, regional, local
- migration: unrestricted, ring, neighborhood
- competition: between subpopulations/strategies
- high level functions to all operators
- different population models supported (global, regional and local model)
- multiple population support
- multiple strategy support (run multiple search strategies beside each other in one optimization run)
- competition between subpopulations possible (provides efficient distribution of computer resources between different subpopulation)
- sophisticated optimization visualization of state and course of the Evolutionary Algorithms, online (during optimization) and off-line (after an optimization run using logged data)
- comfortable monitoring and storing of results (every run can be fully documented in text and data files)
- incorporation of problem specific knowledge (examples included)

D.3 Genetic Algorithm Toolbox

The Genetic Algorithm Toolbox is a module for use with MATLAB that contains software routines for implementing genetic algorithms (GAs) and other evolutionary computing techniques. The Genetic Algorithm Toolbox was developed by Andrew Chipperfield, Carlos Fonseca, Peter Fleming and Hartmut Pohlheim, who are internationally known for their research and applications in this area. The toolbox is a collection of specialised MATLAB functions supporting the development and implementation of genetic and evolutionary algorithms.

Its main features include:

- Support for binary, integer and real-valued representations.
- A wide range of genetic operators.
- High-level entry points to most low-level functions allowing the user greater ease and flexibility in creating GA applications.
- Many variations on the standard GA.
- Support for virtual multiple subpopulations.

Consistent with the open-system approach of other MATLAB toolboxes, the Genetic Algorithm Toolbox is extensible to suit the user's needs. In combination with other MATLAB toolboxes and SIMULINK, the toolbox provides a versatile and powerful environment for exploring and developing genetic algorithms. The Genetic Algorithm Toolbox is available on an unsupported basis for a modest charge.

D.4 Genetic Programming Toolbox for MATLAB

GPLAB is a Genetic Programming toolbox for MATLAB. Most of its functions are used as "plug and play" devices, making it a versatile and easily extendable tool, as long as the users have minimum knowledge of the MATLAB programming environment.

Some of the features of GPLAB include:

- 3 modes of tree initialization (Full, Grow, Ramped Half-and-Half) +3 variations on these
- several pre-made functions and terminals for building trees
- dynamic limits on tree depth or size (optional)
- resource-limited GP (variable size populations) (optional)
- dynamic populations (variable size populations) (optional)
- 4 genetic operators (crossover, mutation, swap mutation, shrink mutation)
- configurable automatic adaptation of operator probabilities (optional)
- steady-state + generational + batch modes, with fuzzy frontiers between them
- 5 sampling methods (Roulette, SUS, Tournament, Lexicographic Parsimony Pressure Tournament, Double Tournament)
- 3 modes of calculating the expected number of offspring (absolute +2 ranking methods)
- 2 methods for reading input files and for calculating fitness (symbolic regression and parity problems + artificial ant problems)
- runtime cross-validation of the best individual of the run (optional)
- offline cross-validation or prediction of results by any individual (optional)
- 4 levels of elitism
- configurable stop conditions
- saving of results to files (5 frequency modes, optional)
- 3 modes of runtime textual output
- runtime graphical output (4 plots, optional)
- offline graphical output (5 functions, optional)
- runtime measurement of population diversity (2 measures, optional)
- runtime measurement of average tree level, number of nodes, number of introns, tree fill rate (optional)
- 4 demonstration functions (symbolic regression, parity, artificial ant, multiplexer)

No matter how long this list could be, the best feature of GPLAB will always be the "plug and play" philosophy. Any alternative (or collective) function built by the user will be readily accepted by the toolbox, as long as it conforms to the rules pertaining the module in question. Also, there are no incompatibilities between functions and parameters, meaning the user can use any combination of them, even when the user uses their own functions.

GPLAB does not implement:

- multiple subpopulations
- automatically-defined functions

Appendix – E
Commercial Software Packages

This appendix gives a list of all known software packages related to Evolutionary Computation that is available to the users.

E.1 ActiveGA

ActiveGA is an activeX (OLE) control that uses a Genetic Algorithm to find a solution for a given problem. For example, the user can insert an ActiveGA control into Microsoft Excel 97 and have it optimize the worksheet.

Features include:

- Optimization Mode: Minimize, Maximize or Closest To
- Selection Mode: Tournament, Roulette Wheel
- User defined population size, mutation rate and other parameters
- Event driven, cancelable iteration
- Invisible at run time
- Excel 97, Visual Basic, Visual C^{++} samples

E.2 EnGENEer

Logica Cambridge Ltd. developed EnGENEer as an in-house Genetic Algorithm environment to assist the development of GA applications on a wide range of domains. The software was written in C and runs under Unix as part of a consultancy and systems package. It supports both interactive (X-Windows) and batch (command-line) modes of operation.

EnGENEer provides a number of flexible mechanisms which allow the developer to rapidly bring the power of GAs to bear on new problem domains. Starting

with the Genetic Description Language, the developer can describe, at high level, the structure of the "genetic material" used. The language supports discrete genes with user defined cardinality and includes features such as multiple chromosomes models, multiple species models and non-evolvable parsing symbols which can be used for decoding complex genetic material.

The user also has available a descriptive high level language, the Evolutionary Model Language. It allows the description of the GA type used in terms of configurable options including: population size, population structure and source, selection method, crossover and mutation type and probability, inversion, dispersal method, and number of offspring per generation.

Both the Genetic Description Language and the Evolutionary Model Language are fully supported within the interactive interface (including online help system) and can be defined either "on the fly" or loaded from audit files which are automatically created during a GA run. Monitoring of GA progress is provided via both graphical tools and automatic storage of results (at user defined intervals). This allows the user to restart EnGENEer from any point in a run, by loading both the population at that time and the evolutionary model that was being used.

Connecting EnGENEer to different problem domains is achieved by specifying the name of the program used to evaluate the problem specific fitness function and constructing a simple parsing routine to interpret the genetic material. A library of standard interpretation routines are also provided for commonly used representation schemes such as gray-coding, permutations, etc. The fitness evaluation can then be run as either a slave process to the GA or via a standard handshaking routines. Better still, it can be run on either the machine hosting the EnGENEer or on any sequential or parallel hardware capable of connecting to a Unix machine.

E.3 EvoFrame

EvoFrame is to Evolution Strategies what MicroGA is to Genetic Algorithms, a toolkit for application development incorporating ES as the optimization engine.

EvoFrame is an object oriented implemented programming tool for evolution strategies (Rechenberg/Schwefel, Germany) for easy implementation and solution of numerical and combinatorical problems. EvoFrame gives freedom of implementing every byte of the optimization principle and its user interface.

EvoFrame is available as Version 2.0 in Borland-Pascal 7.0 and Turbo-Vision for PC's and as Version 1.0 in C^{++} for Apple Macintosh using MPW and MacApp. Both implementations allow full typed implementation, i.e. *no more* translation from problem specific format to an optimization specific one. A prototyping tool (cf REALizer) exists for both platforms too.

EvoFrame allows pseudoparallel optimization of many problems at once and the user can switch optimization parameters and internal methods (i.e. quality function etc.) during runtime and during optimization cycle. Both tools can be modified or extended by overloading existing methods for experimental use. They are developed continously in correlation to new research results.

E.4 REALizer

REALizer is a tool for rapid prototyping of EvoFrame applications. It's an override of the corresponding framework which is prepared to optimize using a vector of real numbers. All methods for standard evolution and file handling, etc. are ready implemented. The remaining work for the user is to define a constant for the problem size, fill in the quality function and start the optimization process.

E.5 Evolver

Evolver is a Genetic Algorithm package for Windows. Beginners can use the Excel add-in to model and solve problems from within Excel. Advanced users can use the included Evolver API to build custom applications that access any of the six different genetic algorithms. Evolver can be customized and users can monitor progress in real-time graphs, or change parameters through the included Evolver-Watcher program.

E.6 FlexTool

FlexTool(GA) is a modular software tool which provides an environment for applying GA to diverse domains with minimum user interaction and design iteration.

Version M2.2 is the MATLAB version which provides a total GA based design and development environment in MATLAB. MATLAB provides us with an interactive computation intensive environment. The high level, user friendly programming language combined with built-in functions to handle matrix algebra, Fourier series, and complex valued functions provides the power for large scale number crunching.

The GA objects are provided as .m files. FlexTool(GA) Version M2.2 is designed with emphasis on modularity, flexibility, user friendliness, environment transparency, upgradability, and reliability. The design is engineered to evolve complex, robust models by drawing on the power of MATLAB.

FlexTool(GA) Version M2.2 Features:

Building Block	: Upgrade to EFM or ENM or CI within one year
Niching module	: to identify multiple solutions
Clustering module	: Use separately or with Niching module
Optimization	: Single and Multiple Objectives
Flex-GA	: Very fast proprietary learning algorithm
GA	: Modular, User Friendly, and System Transparent
GUI	: Easy to use, user friendly
Help	: Online
Tutorial	: Hands-on tutorial, application guidelines

Parameter Settings	: Default parameter settings for the novice
General	: Statistics, figures, and data collection
Compatibility	: FlexTool product suite
GA options	: generational, steady state, micro, Flex-GA
Coding schemes	: include binary, logarithmic, real
Selection	: tournament, roulette wheel, ranking
Crossover	: include 1, 2, multiple point crossover
Compatible to	: FlexTool(GA) M1.1 Genetic Algorithms Toolbox

The FlexTool product suite includes various soft computing building blocks:

CI: Computational Intelligence http://www.flextool.com/ftci.html
EFM: Evolutionary Fuzzy Modeling http://www.flextool.com/ftefm.html
ENM: Evolutionary Neuro Modeling http://www.flextool.com/ftenm.html
FS : Fuzzy Systems http://www.flextool.com/ftfs.html
EA : Evolutionary Algorithms http://www.flextool.com/ftga.html
NN : Neural Networks http://www.flextool.com/ftnn.html

E.7 GAME

GAME (GA Manipulation Environment) aims to demonstrate GA applications and build a suitable programming environment. GAME is being developed as part of the PAPAGENA project of the European Community's Esprit III initiative.

GAME is now in version 2.01. This version is still able to run only sequential GAs, but version 3.0 will handle parallel GAs as well.

The project yet only produced a Borland C^{++} 3.x version, so far. It is intended to distribute a version for UNIX/GNU C^{++} as well, when some compatibility issues concerning C^{++} "standards" have been resolved. Afterward a UNIX version will be released, but this will be only happen after the release of PC version 3.0.

E.8 GeneHunter

GeneHunter from Ward Systems runs on a PC under Windows. It is callable from Microsoft Excel 5 spreadsheets, and accessible via function calls in a dynamic link library. The DLL is designed especially for Visual Basic, but runs with other languages which call DLLs under Windows 3.1 such as Visual C^{++}. 16- and 32-bit versions are available. GeneHunter can also integrate with Ward's neural network software.

E.9 Generator

Generator is a Genetic Algorithm package designed to interact with Microsoft Excel for Windows. Users are able to define and solve problems using Excel formulas,

tables and functions. Fitness is easily defined as an Excel formula or optionally a macro. Progress can be monitored using Generator's real-time fitness graph and status window as well as user-defined Excel graphs. Generator can be paused at any time to allow adjustment of any of the parameters and then resumed.

Generator Features:

- Multiple gene types: integer, real and permutation.
- Combined roulette-wheel and elitist selection method.
- Elitism is optional and adjustable.
- None, two-point, and a proprietary permutation crossover.
- Random, Random Hillclimb and Directional Hillclimb mutation methods.
- Special hillclimbing features to find solutions faster.
- Fitness goal: maximize, minimize or seek value.
- Convergence: duplicates not allowed.
- Real-Time alteration of parameters relating to crossover, mutation, population, etc.
- Real-Time progress graph of Best, Worst and Median fitness.
- Fitness defined using an Excel formula or macro.

The parameters available to the user include mutation probability for population and genes, control of mutation limit per gene, control of hillclimbing, population size, elite group size, recombination method, and mutation technique.

Connecting generator to problems defined on the Excel spreadsheet is achieved by first specifying the spreadsheet locations of the gene group cells and their type, and lastly, the location of the formula used to evaluate the problem-specific fitness function. Generator requires at least a 386 IBM compatible PC with 2 MB of RAM, Windows 3.0 (or later) and Microsoft Excel 4.0 (or later).

E.10 Genetic Server and Genetic Library

Genetic Server and *Genetic Library* are tools that allow programmers to embed Genetic Algorithms into their own applications. Both products provide a flexible yet intuitive API for genetic algorithm design. Genetic Server is an ActiveX component designed to be used within a Visual Basic (or VBA) application and Genetic Library is a C^{++} library designed to be used within a Visual C^{++} application.

Features include:

- Data types: Binary, Integer, and Real
- Progression types: Generational, Steady State
- Selection operators: Roulette (Fitness or Rank), Tournament, Top Percent, Best, and Random
- Crossover operators: One Point, Two Point, Uniform, Arithmetic, and Heuristic
- Mutation operators: Flip Bit, Boundary, Non-Uniform, Uniform, and Gaussian

- Termination Methods: Generation Number, Evolution Time, Fitness Threshold, Fitness Convergence, Population Convergence, and Gene Convergence
- User-defined selection, crossover, and mutation operators (Genetic Library only)

E.11 MicroGA

MicroGA is a powerful and flexible new tool which allows programmers to integrate GAs into their software quickly and easily. It is an object-oriented C^{++} framework that comes with full source code and documentation as well as three sample applications. Also included is the Galapagos code generator which allows users to create complete applications interactively without writing any C^{++} code, and a sample MacApp interface.

MicroGA is available for Macintosh II or higher with MPW and a C^{++} compiler, and also in a Microsoft Windows version for PC compatibles.

Galapagos is a tool for use with Emergent Behavior's MicroGA Toolkit. It allows a user to define a function and set of constraints for a problem that the user wants to solve using the GA. Galapagos then generates a complete C^{++} program using the information supplied. Then all the user has to do is to compile these files, using either Turbo/Borland C^{++} (PC, MS Windows), or MPW and C^{++} compiler (Macintosh), and link the resulting code to the MicroGA library. Then just run the program.

E.12 Omega

The Omega Predictive Modeling System, marketed by KiQ Limited, is a powerful approach to developing predictive models. It exploits advanced GA techniques to create a tool which is "flexible, powerful, informative and straightforward to use". Omega is geared to the financial domain, with applications in Direct Marketing, Insurance, Investigations and Credit Management. The environment offers facilities for automatic handling of data; business, statistical or custom measures of performance, simple and complex profit modeling, validation sample tests, advanced confidence tests, real time graphics, and optional control over the internal GA.

E.13 OOGA

OOGA (Object-Oriented GA) is a Genetic Algorithm designed for industrial use. It includes examples accompanying the tutorial in the companion "Handbook of Genetic Algorithms". OOGA is designed such that each of the techniques employed by a GA is an object that may be modified, displayed or replaced in object-oriented fashion. OOGA is especially well-suited for individuals wishing to modify the basic GA techniques or tailor them to new domains.

E.14 OptiGA

OptiGA for VB is an ActiveX control (OCX) for the implementation of Genetic Algorithms. It is described by the author, Elad Salomons, as follows:

No matter what the nature of the optimization problem might be, optiGA is a generic control that will perform the genetic run for the user. With very little coding needed, the user can be up and running in no time. Just define the variables (binary, real or integers), code the fitness function. On the other hand, the user can override optiGA's default parameters and select from several of reproduction operators such as: selection methods, crossover methods, mutation methods and many controlling parameters.

OptiGA was written in "Visual Basic" and can be used with VB and all supporting environments.

E.15 PC-Beagle

PC-Beagle is a rule-finder program for PCs which examines a database of examples and uses machine-learning techniques to create a set of decision rules for classifying those examples, thus turning data into knowledge. The system contains six major components, one of which (HERB – the "Heuristic Evolutionary Rule Breeder") uses GA techniques to generate rules by natural selection.

E.16 XpertRule GenAsys

XpertRule GenAsys is an expert system shell with embedded GENETIC ALGORITHM marketed by Attar Software. Targeted to solve scheduling and design applications, this system combines the power of genetic algorithms in evolving solutions with the power of rule-based programming in analyzing the effectiveness of solutions. Rule-based programming can also be used to generate the initial POPULATION for the genetic algorithm and for post-optimization planning. Some examples of design and scheduling problems which can be solved by this system include: OPTIMIZATION of design parameters in electronic and avionic industries, route optimization in the distribution sector, production scheduling in manufacturing, etc.

E.17 XYpe

XYpe (The GA Engine) is a commercial GA application and development package for the Apple Macintosh. Its standard user interface allows the user to design chromosomes, set attributes of the genetic engine and graphically display its progress. The development package provides a set of Think C libraries and includes files for

the design of new GA applications. XYpe supports adaptive operator weights and mixtures of alpha, binary, gray, ordering and real number codings.

E.18 Evolution Machine

The Evolution Machine (EM) is universally applicable to continuous (real-coded) optimization problems. In the EM the fundamental Evolutionary Algorithms (Genetic Algorithms and Evolution Strategies) are coded, and some of the approaches to evolutionary search are added.

The EM includes extensive menu techniques with:

- Default parameter setting for unexperienced users.
- Well-defined entries for EM-control by freaks of the EM, who want to leave the standard process control.
- Data processing for repeated runs (with or without change of the strategy parameters).
- Graphical presentation of results: online presentation of the EVOLUTION progress, one-, two- and three-dimensional graphic output to analyse the FITNESS function and the evolution process.
- Integration of calling MS-DOS utilities (Turbo C).

E.19 Evolutionary Objects

EO (Evolutionary Objects) is a C^{++} library written and designed to allow a variety of evolutionary algorithms to be constructed easily. It is intended to be an "Open source" effort to create the definitive EC library. It has: a mailing list, anon-CVS access, frequent snapshots and other features.

E.20 GAC, GAL

GAC is a GA written in C. GAL is the Common Lisp version. They are similar in spirit to John Grefenstette's Genesis, but they don't have all the nice bells and whistles. Both versions currently run on Sun workstations.

E.21 GAGA

GAGA (GA for General Application) is a self-contained, re-entrant procedure which is suitable for the minimization of many "difficult" cost functions. Originally written in Pascal by Ian Poole, it was rewritten in C by Jon Crowcroft.

E.22 GAGS

GAGS (Genetic Algorithms from Granada, Spain) are a library and companion programs written and designed to take the heat out of designing a Genetic Algorithm. It features a class library for genetic algorithm programming, but, from the user point of view, is a genetic algorithm application generator. Just write the function required to optimize, and GAGS surrounds it with enough code to have a genetic algorithm up and running, compiles it, and runs it. GAGS is written in C^{++}, so that it can be compiled in any platform running this GNU utility.

GAGS includes:

- Steady-state, roulette-wheel, tournament and elitist selection.
- Fitness evaluation using training files.
- Graphics output through gnuplot.
- Uniform and 2-point crossover, and bit-flip and gene-transposition mutation.
- Variable length chromosomes and related operators.

E.23 GAlib

GAlib is a C^{++} library that provides the application programmer with a set of Genetic Algorithm objects. With GAlib the user can add GA optimization to his program using any data representation and standard or custom selection, crossover, mutation, scaling, and replacement, and termination methods.

E.24 GA Workbench

A mouse-driven interactive GA demonstration program aimed at people wishing to show GAs in action on simple function optimizations and to help newcomers understand how GAs operate.

Features include:

- problem functions drawn on screen using mouse
- run-time plots of GA population distribution
- peak and average fitness
- useful population statistics displayed numerically

E.25 Genesis

Genesis is a generational GA system written in C by John Grefenstette. As the first widely available GA program Genesis has been very influential in stimulating the

use of GAs, and several other GA packages are based on it. Genesis is available together with OOGA.

E.26 Genie

Genie is a GA-based modeling/forecasting system that is used for long-term planning. One can construct a model of an environment and then view the forecasts of how that environment will evolve into the future. It is then possible to alter the future picture of the environment so as to construct a picture of a desired future. The GA is then employed to suggest changes to the existing environment so as to cause the desired future to come about.

E.27 XGenetic

XGenetic is an ActiveX control for the implementation of a Genetic Algorithm in any language that accepts ActiveX interfaces. Such languages include, but are not limited to: Visual Basic, Visual C^{++}, Delphi, etc. Written in Visual Basic 6.0, XGenetic is flexible in implementation to allow the user to easily define the parameters for their particular scenario, be it forecasting, scheduling, or the myriad of other uses for the genetic algorithm.

Features include:

- Data Types: Bit, Integer, Real
- Selection Operators: Roulette, Tournament, Stochastic Universal Sampling, Truncation, Random
- Crossover Operators: N-Point (1 point, 2 point, 3 point, etc), Uniform, Arithmetic
- Mutation Operators: Uniform, Boundary

Appendix – F
GA Source Codes in 'C' Language

Codes in 'c' language also serve to explain how simple a GA can be implemented for a searching problem. A few examples of GA implementation in 'c' language are discussed in this appendix. This appendix also elaborates on the steps used in MATLAB to plot the output of GA implemented in 'c' language.

F.1 A "Hello World" Genetic Algorithm Example

Problem Description:

The code described in this section evolves a random group of letters into the phrase "Hello World."

Solution:

The GA code used to implement this problem has the following characteristics:

Population Size: 2048
Mutation (%): 25
Elitism (%): 10

There are a few things to note about the code as well. Firstly, the size of the strings in the GA population is fixed to be the same size as the target string. This just makes the code easier to understand. Secondly, the class vector is used to hold the GA details - this also makes the sorting very efficient as simple to implement. Finally, the program uses two arrays to hold the population - one for the current population, and another as the buffer to contain the next generation. At the end of each generation, the pointers are swapped round to keep things fast.

CPP Code:

This code serves as a decent look at how simple a GA can be implemented for a searching problem. Hopefully the code is ANSI compatible.

```
# include <iostream>            // for cout etc.
# include <vector>              // for vector class
# include <string>              // for string class
# include <algorithm>           // for sort algorithm
# include <time.h>              // for random seed
# include <math.h>              // for abs()
# define GA_POPSIZE      2048     // ga population size
# define GA_MAXITER      16384   // maximum iterations
# define GA_ELITRATE     0.10f    // elitism rate
# define GA_MUTATIONRATE 0.25f    // mutation rate
# define GA_MUTATION     RAND_MAX * GA_MUTATIONRATE
# define GA_TARGET       std::string("Hello world!")
using namespace std;     // polluting global namespace
struct ga_struct
{
       string str;                // the string
       unsigned int fitness;      // its fitness
};
typedef vector<ga_struct> ga_vector; // for brevity
void init_population(ga_vector &population,
                         ga_vector &buffer)
{
       int tsize = GA_TARGET.size();
       for (int i=0; i<GA_POPSIZE; i++) {
              ga_struct citizen;
              citizen.fitness = 0;
              citizen.str.erase();
              for (int j=0; j<tsize; j++)
                     citizen.str += (rand() % 90) + 32;
              population.push_back(citizen);
       }
       buffer.resize(GA_POPSIZE);
}
```

// Basically, the fitness function goes through each member of the population and compares it with the target string. It adds up the differences between the characters and uses the cumulative sum as the fitness value (therefore, the lower the value, the better).

```
void calc_fitness(ga_vector &population)
{
```

```
        string target = GA_TARGET;
        int tsize = target.size();
        unsigned int fitness;

        for (int i=0; i<GA_POPSIZE; i++) {
              fitness = 0;
              for (int j=0; j<tsize; j++) {
                 fitness += abs(int(population[i].str[j]
                              - target[j]));
              }
              population[i].fitness = fitness;
        }
}
bool fitness_sort(ga_struct x, ga_struct y)
{ return (x.fitness < y.fitness); }
inline void sort_by_fitness(ga_vector &population)
{ sort(population.begin(), population.end(),
  fitness_sort);
}

void elitism(ga_vector &population,
                       ga_vector &buffer, int esize )
{
       for (int i=0; i<esize; i++) {
            buffer[i].str = population[i].str;
            buffer[i].fitness = population[i].fitness;
       }
}

void mutate(ga_struct &member)
{
       int tsize = GA_TARGET.size();
       int ipos = rand() % tsize;
       int delta = (rand() % 90) + 32;

       member.str[ipos] = ((member.str[ipos] + delta)
       % 122);
}
void mate(ga_vector &population, ga_vector &buffer)
{
       int esize = GA_POPSIZE * GA_ELITRATE;
       int tsize = GA_TARGET.size(), spos, i1, i2;
       elitism(population, buffer, esize);
       // Mate the rest
       for (int i=esize; i<GA_POPSIZE; i++) {
```

```
                i1 = rand() % (GA_POPSIZE / 2);
                i2 = rand() % (GA_POPSIZE / 2);
                spos = rand() % tsize;

                buffer[i].str = population[i1].str.
                substr(0,
                spos) + population[i2].str.substr(spos,
                esize --
                spos);
                if (rand() < GA_MUTATION) mutate(buffer
                [i]);
        }
}
```

// To print the output

```
inline void print_best(ga_vector &gav)
{ cout << "Best: " << gav[0].str << " (" << gav[0].
  fitness << ")" << endl; }
inline void swap(ga_vector *&population,
ga_vector *&buffer)
{ ga_vector *temp = population; population = buffer;
  buffer = temp; }

int main()
{
        srand(unsigned(time(NULL)));
        ga_vector pop_alpha, pop_beta;
        ga_vector *population, *buffer;

        init_population(pop_alpha, pop_beta);
        population = &pop_alpha;
        buffer = &pop_beta;

        for (int i=0; i<GA_MAXITER; i++) {
                // calculate fitness
                calc_fitness(*population);

                // sort them
                sort_by_fitness(*population);
                // print the best one
                print_best(*population);
                if ((*population)[0].fitness == 0) break;
                // mate the population together
                mate(*population, *buffer);
```

```
                // swap buffers
                swap(population, buffer);
        }
        return 0;
}
```

Output:

The output of the above program with a population size of 2048 and 0.25 mutation is shown below. Here the best population member and the fitness are displayed

```
Best: IQQte=Ygqem# (152)
Best: Crmt`!qrya+6 (148)
Best: 8ufxp+Rigfm* (140)
Best: b`hpf"woljh[ (120)
Best: b`hpf"woljh4 (81)
Best: b`hpf"woljh" (63)
Best: Kdoit!wnsk_! (24)
Best: Kdoit!wnsk_! (24)
Best: Idoit!wnsk_! (22)
Best: Idoit!wnsk_! (22)
Best: Idoit!wnsk_! (22)
Best: Idoit!wnsk_! (22)
Best: Ifknm!vkrlf? (17)
Best: Ifknm!vkrlf? (17)
Best: Gfnio!wnskd$ (14)
Best: Ffnjo!wnskd$ (14)
Best: Hflio!wnskd$ (11)
Best: Hflio!wnskd$ (11)
Best: Kflkn wosld" (8)
Best: Ifmmo workd" (6)
Best: Hfljo wosld" (5)
Best: Hflmo workd" (4)
Best: Hflmo workd" (4)
Best: Hflmo workd" (4)
Best: Iflmo world! (3)
Best: Iflmo world! (3)
Best: Hflmo world! (2)
Best: Hflmo world! (2)
Best: Hflko world! (2)
Best: Hflko world! (2)
Best: Hdllo world! (1)
Best: Hfllo world! (1)
Best: Hfllo world! (1)
```

```
Best: Helko world! (1)
Best: Hfllo world! (1)
Best: Hfllo world! (1)
Best: Hello world! (0)
```

Conclusion:

Thus this simple GA code serves as how to implement a code for a searching problem.

F.2 Test Function Using sin and cos

In this section, a simple genetic algorithm in C# is produced. It will not be multi-threaded, nor will it contain exotic operators or convergence criteria (*i.e*. a condition where many of the solutions found are very similar). It will simply demonstrate a genetic algorithm in managed code, taking advantage of some of the features of the runtime.

Introduction

A genetic algorithm is an optimization technique that relies on parallels with nature. It can tackle a variety of optimization techniques provided that they can be parameterized in such a way that a solution to the problem provides measure of how accurate the solution found by the algorithm is. This measure is defined as fitness.

Genetic algorithms were first conceived in early 1970's (Holland, 1975). The initial idea came from observing how the evolution of biological creatures derives from their constituent DNA and chromosomes. In this sense a simple analogy can be made with a mathematical problem made up of many parameters. Each parameter can take the place of a chromosome in the mathematical analogy of a real chemical sequence.

In nature, evolution is carried out by a process of selection typified by the expression survival of the fittest. In order to select an individual, a population of such individuals has to be chosen to produce a new generation of individuals.

For any problem that is to be solved, some measure of the goodness of the solution is required, i.e. fitness, often a χ^2 (chi-squared) measure, i.e. the better the solution, the higher the fitness returned from out function. The less fit the solutions are, the less likely that they are to survive to a successive population. By employing such a technique, the algorithm can reduce the number of possible solutions that it examines.

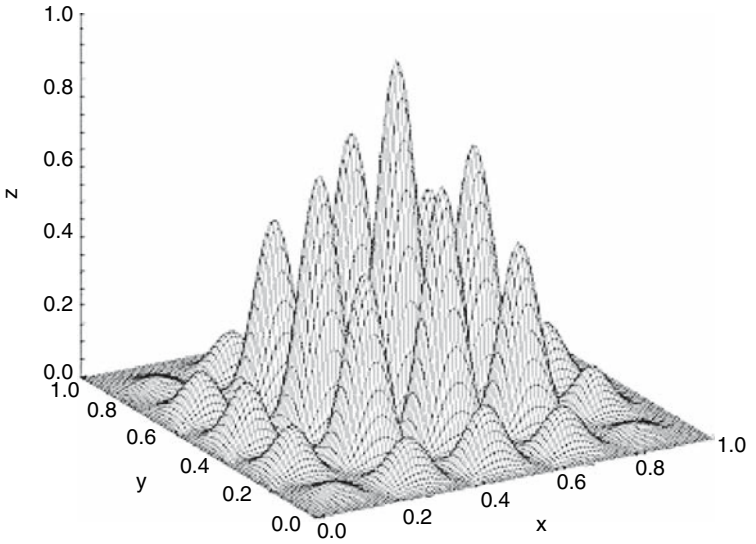


Fig. F2.1 Plot of the problem

Many problems are internally represented in binary by various genetic algorithms. Here a decimal representation is considered. The internal representation of a genetic algorithm does not actually matter provided the implementation is thorough.

The Problem

In this example code, a test function that uses sin and cos is used to produce the plot shown in Figure F2.1:

The optimal solution for this problem is (0.5,0.5), i.e. the highest peak. This example is chosen to demonstrate how a genetic algorithm is not fooled by the surrounding local maxima (i.e. the high peaks).

Test Harness

We start by declaring a new genetic algorithm:

```
GA ga = new GA(0.8,0.05,100,2000,2);
ga.FitnessFunction = new GAFunction(theActualFunction);
```

where the arguments are the crossover rate, mutation rate, population size, number of generations, and number of parameters that we are solving for. We declare the FitnessFunction property as:

```
public delegate double GAFunction(double[] values);

public class GA
{
   static private GAFunction getFitness;
   public GAFunction FitnessFunction {
     // etc.
   };
   // etc.
}
```

This then enables us to declare the fitness function same as the delegate function:

```
public static double theActualFunction(double[]
  values)
{
  if (values.GetLength(0) != 2)
  throw new ArgumentOutOfRangeException("should only
  have 2 args");
   double x = values[0];
   double y = values[1];
   double n = 9;
   double f1 = Math.Pow(15*x*y*(1-x)*(1-y)
               *Math.Sin(n*Math.PI*x)
               *Math.Sin(n*Math.PI*y),2);
   return f1;
}
```

which is therefore accepted by the algorithm. The genetic algorithm is then set to run using:

```
ga.Go();
```

The genetic algorithm will now run for the supplied number of generations.

The Algorithm

The algorithm code contains two simple classes, GA and Genome, plus a helper class GenomeComparer.

The Genome class can be thought of as a simple container. The underlying structure is an array of doubles within the range of 0 to 1. The user is expected to take these values and scale them to whatever values they require. Since mutation occurs on the genome, the Mutate method is found in this class. The Crossover operator requires access to the private data of the Genome, so it is also a member function

which takes a second Genome, and outputs two child Genome objects. The fitness of a particular genome is also stored within the Genome object. There are some additional helper functions that maybe found in the code itself.

The GA class does all the work. The genetic algorithm consists of the following basic steps:

1. Create a new population
2. Select two individuals from the population weighting towards the individual that represents the best solution so far.
3. 'Breed' them to produce children.
4. If we don't have enough children for a new population return to step 2.
5. Replace old population with new.
6. If we have not produced enough generations return to step 2.
7. Completed.

When selecting individuals to breed, we use what is called the Roulette wheel method. This is where fitter individuals have a larger proportion of the 'wheel' and are more likely to be chosen. We chose to store the finesses cumulatively in System.Collections.ArrayList as it had some nice features like sorting. Unfortunately, its binary search method was only for exact values, so we had to implement the following work around:

```
mid = (last - first)/2;
// ArrayList's BinarySearch is for exact values only
// so do this by hand.
while (idx == -1 && first <= last)
{
  if (randomFitness < (double)m_fitnessTable[mid])
  {
     last = mid;
  }
  else if (randomFitness > (double)m_fitnessTable[mid])
  {
     first = mid;
  }
  mid = (first + last)/2;
  // lies between i and i+1
  if ((last - first) == 1)
      idx = last;
}
```

The GenomeComparer class inherits from the IComparer interface. Each generation is stored in a System.Collections.ArrayList, and we wish to sort each generation in order of fitness. We therefore need to implement this interface as follows:

```
public sealed class GenomeComparer : IComparer
 {
     public GenomeComparer()
     {
     }
     public int Compare( object x, object y)
     {
         if ( !(x is Genome) || !(y is Genome))
            throw new ArgumentException("Not of type
            Genome");
         if (((Genome) x).Fitness > ((Genome) y).
            Fitness)
            return 1;
         else if (((Genome) x).Fitness == ((Genome) y).
            Fitness)
            return 0;
         else
            return -1;
    }
}
```

Note that we need to explicitly cast the ArrayList elements back to a Genome type. We also make the class sealed as there is no point inheriting from it.

Results

With this simple example we know that the optimal solution is at (0.5, 0.5), and we find after 250 generations we find a solution extremely close to this (within 4 significant figures). The progress of the GA can be seen in Figure F2.2:

Conclusion

A genetic algorithm does not provide a magic bullet solution to all minimization/maximization problems. In many cases other algorithms are faster and more practical. However for problems with a large parameter space and where the problem itself can be easily specified, it can be an appropriate solution.

Some of the features of C# are:

- The System.Collections.ArrayList container only does shallow copies.
- The System.Collections.ArrayList container's binary search only works for exact values.

- An obvious thing, but simply defining a variable, doesn't necessarily assign it a value.
- Implementing the IComparer interface is fairly trivial (see GenomeComparer class).

Further improvements to the algorithm can be made by implementing all sorts of weird and wonderful operators.

F.3 Using Matlab to Plot Data Generated by C Language

/* The .c file may be compiled using the format:

```
gcc filename.c -o exefile -lm
```

Running this program creates a file named "outvar.dat". The data for a cosine and sine function is calculated and saved as three columns of numbers.

To plot the data within MATLAB, the user may use the following from a MATLAB window:

```
>> load outvar.dat
>> x = outvar(:,1);
>> y = outvar(:,2);
>> z = outvar(:,3);
>> plot(x,y)
```

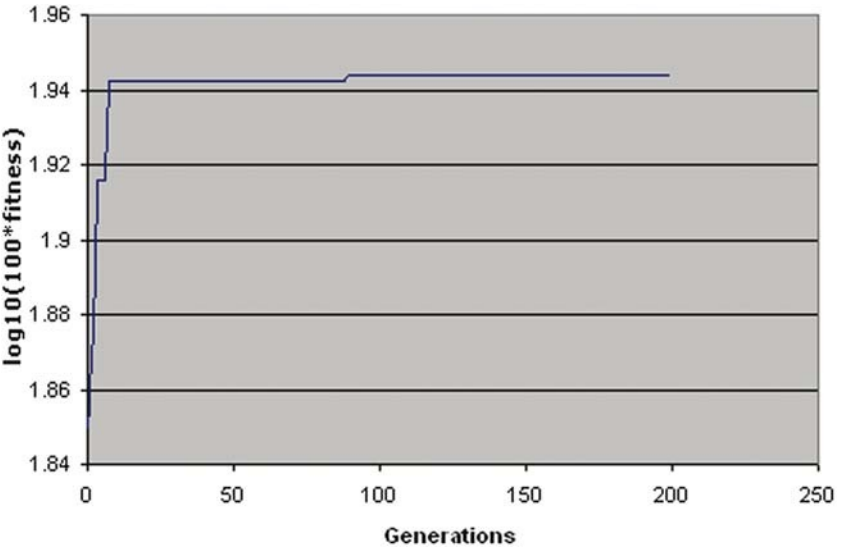


Fig. F2.2 Progress of the GA

```
>> hold on
>> plot(x,z)
>> hold off
*/
#include <stdio.h>        /* standard Input/Output */
#include <stdlib.h>       /* standard library */
#include <math.h>         /* math library */
main()
{
  double x,y,z;            /* variables used */
  FILE *fp;                /* file pointer */
/* Check to see if there are any file errors. */
/* This will either create a new file OUTVAR.DAT, or
*/
/* write over an existing version of the file. */
  if ( (fp = fopen("outvar.dat", "w")) == NULL)
  {
     printf("Cant open OUTVAR.DAT \n");
     exit(1);
  }
/* This loop is used to save y=cos(x).*/
  for(x=0; x<=10; x += 0.1)
  {
     y = cos(x);
     z = sin(x);
     fprintf(fp, "%f %f %f",x,y,z); /* save the x
     and y
values  */
     fprintf(fp, "\n"); /* create a line break */
  }
    fclose(fp); /* close the file */
}
```

For other High Level Languages: If the user want to use Fortran, Pascal, Basic, or some other language and MATLAB to generate plots the user can proceed in a similar manner as that was done with C. Any space delimited ASCII data file can be read by MATLAB (i.e. columns of data seperated by any number of "blank" spaces). Hence, if the user uses any high level language simply send the data into an ASCII file and use the MATLAB commands given above.

Appendix – G
EC Class/Code Libraries and Software Kits

Evolutionary computing is actually a broad term for a vast array of programming techniques, including genetic algorithms, complex adaptive systems, evolutionary programming, etc. The main thrust of all these techniques is the idea of evolution. The idea that a program can be written that will *evolve* toward a certain goal. This goal can be anything from solving some engineering problem to winning a game. Some of the libraries of code and software kits for evolutionary computing are discussed in this appendix.

G.1 EC Class/Code Libraries

These are libraries of code or classes for use in programming within the evolutionary computation field. They are not meant as stand alone applications, but rather as tools for building user's own applications.

ANNEvolve

Web site: annevolve.sourceforge.net

A collection of programs using evolved artificial neural networks to solve a series of problems. The long term goal of the project is to advance the level of understanding about simulated evolution as a means to configure and optimize Artificial Neural Nets (ANNs). The medium term goal is to apply methods to a series of interesting problems such as sail boat piloting and playing the game NIM.

A secondary goal is educational in nature. We attempt to write software with ample explanation, not just for the user, but for the engineer/programmer/scientist who wants to understand the innermost detail. All of the source code is freely available to anyone to use without restriction.

All of the ANNEvolve software is implemented in C and Python.

daga

Web site: garage.cps.msu.edu/software/daga3.2/

daga is an experimental release of a 2-level genetic algorithm compatible with the GALOPPS GA software. It is a meta-GA which dynamically evolves a population of GAs to solve a problem presented to the lower-level GAs. When multiple GAs (with different operators, parameter settings, etc.) are simultaneously applied to the same problem, the ones showing better performance have a higher probability of surviving and "breeding" to the next macro-generation (i.e., spawning new "daughter"- GAs with characteristics inherited from the parental GA or GAs. In this way, we try to encourage good problem-solving strategies to spread to the whole population of GAs.

dgpf

Web site: dgpf.sourceforge.net

The Distributed Genetic Programming Framework (DGPF) is a scalable Java environment for heuristic, simulation-based search algorithms of any kind and Genetic Algorithms in special. We use the broad foundation of a search algorithms layer to provide a Genetic Programming system which is able to create Turing-complete code.

It's under the LGPL license. It allows the user to use heuristic searches like GA and randomized Hill Climbing for any problem space with just minimal programming effort. Also, the user may distribute all these searches over a network, using the client/server, the peer-to-peer, or even a client/server+ peer-to-peer hybrid distribution scheme. The user can also construct heterogeneous search algorithms where GA cooperates with Hill Climbing without changing any code.

Ease

Web site: www.sprave.com/Ease/Ease.html

Ease - Evolutionary Algorithms Scripting Environment - is an extension to the Tcl scripting language, providing commands to create, modify, and evaluate populations of individuals represented by real number vectors and/or bit strings.

EO

Web site: eodev.sourceforge.net

EO is a templates-based, ANSI-C++ compliant evolutionary computation library. It contains classes for any kind of evolutionary computation (especially genetic al-

gorithms). It is component-based, so that if the user does not find the class required in it, it is very easy to subclass existing abstract or concrete class.

FORTRAN GA

Web site: cuaerospace.com/carroll/ga.html

This program is a FORTRAN version of a genetic algorithm driver. This code initializes a random sample of individuals with different parameters to be optimized using the genetic algorithm approach, i.e. evolution via survival of the fittest. The selection scheme used is tournament selection with a shuffling technique for choosing random pairs for mating. The routine includes binary coding for the individuals, jump mutation, creep mutation, and the option for single-point or uniform crossover. Niching (sharing) and an option for the number of children per pair of parents has been added. More recently, an option for the use of a micro-GA has been added.

GAlib: Matthew's Genetic Algorithms Library

Web Site: lancet.mit.edu/ga/

GAlib contains a set of C++ genetic algorithm objects. The library includes tools for using genetic algorithms to do optimization in any C++ program using any representation and genetic operators. The documentation includes an extensive overview of how to implement a genetic algorithm as well as examples illustrating customizations to the GAlib classes.

GALOPPS

Web site: garage.cps.msu.edu/software/galopps/

GALOPPS is a flexible, generic GA, in 'C'. It was based upon Goldberg's Simple Genetic Algorithm (SGA) architecture, in order to make it easier for users to learn to use and extend.

GALOPPS extends the SGA capabilities several fold:

- A new Graphical User Interface, based on TCL/TK, for Unix users, allowing easy running of GALOPPS 3.2 (single or multiple subpopulations) on one or more processors. GUI writes/reads "standard" GALOPPS input and master files, and displays graphical output (during or after run) of user-selected variables.
- 5 selection methods: roulette wheel, stochastic remainder sampling, tournament selection, stochastic universal sampling, and linear-ranking.
- Random or super uniform initialization of "ordinary" (non-permutation) binary or non-binary chromosomes; random initialization of permutation-based chromosomes; or user-supplied initialization of arbitrary types of chromosomes.

- Binary or non-binary alphabetic fields on value-based chromosomes, including different user-definable field sizes.
- 3 crossovers for value-based representations: 1-pt, 2-pt, and uniform, all of which operate at field boundaries if a non-binary alphabet is used.
- 4 crossovers for order-based reps: PMX, order-based, uniform order-based, and cycle.
- 4 mutations: fast bitwise, multiple-field, swap and random sublist scramble.
- Fitness scaling: linear scaling, Boltzmann scaling, sigma truncation, window scaling, ranking.

GAS

Web site: starship.skyport.net/crew/gandalf

GAS means "Genetic Algorithms Stuff". GAS is freeware. Purpose of GAS is to explore and exploit artificial evolutions. Primary implementation language of GAS is Python. The GAS software package is meant to be a Python framework for applying genetic algorithms. It contains an example application where it is tried to breed Python program strings. This special problem falls into the category of Genetic Programming (GP), and/or Automatic Programming. Nevertheless, GAS tries to be useful for other applications of Genetic Algorithms as well.

GAUL

Web site: gaul.sourceforge.net

The Genetic Algorithm Utility Library (GAUL) is a flexible programming library designed to aid development of applications that require the use of genetic algorithms. Features include:

- Darwinian, Lamarckian or Baldwinian evolutionary schemes.
- Both steady-state and generation-based GAs included.
- The island model of evolution is available.
- Chromosome datatype agnostic. A selection of common chromosome types are built-in.
- Allows user-defined crossover, mutation, selection, adaptation and replacement operators.
- Support for multiple, simultaneously evolved,populations.
- Choice of high-level or low-level interface functions.
- Additional, non-GA, optimisation algorithms are built-in for local optimisation or comparative purposes.
- Trivial to extend using external code via the built-in code hooks.
- May be driven by, or extended by, powerful S-Lang scripts.
- Support for multiprocessor calculations.
- Written using highly portable C code.

GECO

FTP site: common-lisp.net/project/geco/

GECO (Genetic Evolution through Combination of Objects), is an extendible object-oriented tool-box for constructing genetic algorithms (in Lisp). It provides a set of extensible classes and methods designed for generality. Some simple examples are also provided to illustrate the intended use.

Genetic

Web site: packages.qa.debian.org/g/genetic.html

This is a package for genetic algorithms and AI in Python. Genetic can typically solve ANY problem that consists to minimize a function. It also includes several demos/examples, like the TSP (traveling salesman problem).

GPdata

FTP site: ftp.cs.bham.ac.uk/pub/authors/W.B.Langdon/gp-code/

GPdata-3.0.tar.gz (C++) contains a version of Andy Singleton's GP-Quick version 2.1 which has been extensively altered to support:

- Indexed memory operation (cf. teller)
- multi tree programs
- Adfs
- parameter changes without recompilation
- populations partitioned into demes
- (A version of) pareto fitness

This ftp site also contains a small C++ program (ntrees.cc) to calculate the number of different there are of a given length and given function and terminal set.

gpjpp Genetic Programming in Java

Web site: http://www1.cs.columbia.edu/∼evs/ml/hw4.html

gpjpp is a Java package written for doing research in genetic programming. It is a port of the gpc++ kernel written by Adam Fraser and Thomas Weinbrenner. Included in the package are four of Koza's standard examples: the artificial ant, the hopping lawnmower, symbolic regression, and the boolean multiplexer. Here is a partial list of its features:

- graphic output of expression trees
- efficient diversity checking
- Koza's greedy over-selection option for large populations

- extensible GPRun class that encapsulates most details of a genetic programming test
- more robust and efficient streaming code, with automatic checkpoint and restart built into the GPRun class
- an explicit complexity limit that can be set on each GP
- additional configuration variables to allow more testing without recompilation
- support for automatically defined functions (ADFs)
- tournament and fitness proportionate selection
- demetic grouping
- optional steady state population
- subtree crossover
- swap and shrink mutation

jaga

Web site: cs.felk.cvut.cz/∼koutnij/studium/jaga/jaga.html
Simple genetic algorithm package written in Java.

patched lil-gp *

Web site: www.cs.umd.edu/users/seanl/gp/
lil-gp is a generic 'C' genetic programming tool. It was written with a number of goals in mind: speed, ease of use and support for a number of options including:

- Generic 'C' program that runs on UNIX workstations
- Support for multiple population experiments, using arbitrary and user settable topologies for exchange, for a single processor
- lil-gp manipulates trees of function pointers which are allocated in single, large memory blocks for speed and to avoid swapping.

* The patched lil-gp kernel is strongly-typed, with modifications on multithreading, coevolution, and other tweaks and features.

Lithos

Web site: www.esatclear.ie/∼rwallace/lithos.html
Lithos is a stack based evolutionary computation system. Unlike most EC systems, its representation language is computationally complete, while also being faster and more compact than the S-expressions used in genetic programming. The version presented here applies the system to the game of Go, but can be changed to other problems by simply plugging in a different evaluation function. ANSI C source code is provided.

Open BEAGLE

Web site: beagle.gel.ulaval.ca

Open BEAGLE is a C++ evolutionary computation framework. It provides a high-level software environment to do any kind of evolutionary computation, with support for tree-based genetic programming, bit string and real-valued genetic algorithms, evolution strategy, co-evolution, and evolutionary multi-objective optimization.

PGAPack

Web site: www-fp.mcs.anl.gov/CCST/research/reports_pre1998/comp_bio/stalk/pgapack.html

PGAPack (Parallel Genetic Algorithm Library) is a general-purpose, data-structure-neutral, parallel genetic algorithm library. It is intended to provide most capabilities desired in a genetic algorithm library, in an integrated, seamless, and portable manner. Key features are in PGAPack V1.0 include:

- Callable from Fortran or C.
- Runs on uniprocessors, parallel computers, and workstation networks.
- Binary-, integer-, real-, and character-valued native data types.
- Full extensibility to support custom operators and new data types.
- Easy-to-use interface for novice and application users.
- Multiple levels of access for expert users.
- Parameterized population replacement.
- Multiple crossover, mutation, and selection operators.
- Easy integration of hill-climbing heuristics.
- Extensive debugging facilities.
- Large set of example problems.
- Detailed users guide.

PIPE

Web site: www.idsia.ch/∼rafal/research.html

Probabilistic Incremental Program Evolution (PIPE) is a novel technique for automatic program synthesis. The software is written in C. It includes features such as

- Easy to install (comes with an automatic installation tool).
- Easy to use: setting up PIPE_V1.0 for different problems requires a minimal amount of programming. User-written, application-independent program parts can easily be reused.
- Efficient: PIPE_V1.0 has been tuned to speed up performance.

- Portable: comes with source code (optimized for SunOS 5.5.1).
- Extensively documented and contains three example applications.
- Supports statistical evaluations: it facilitates running multiple experiments and collecting results in output files.
- Includes testing tool for testing generalization of evolved programs.
- Supports floating point and integer arithmetic.
- Has extensive output features.
- For lil-gp users: Problems set up for lil-gp 1.0 can be easily ported to PIPE_v1.0. The testing tool can also be used to process programs evolved by lil-gp 1.0.

pygene

Web site: www.freenet.org.nz/python/pygene/

pygene is a simple and easily understandable library for genetic algorithms and genetic programming in python. Includes examples such as the travelling salesman problem.

Sugal

Web site: www.trajan-software.demon.co.uk/sugal.htm

Sugal [soo-gall] is the SUnderland Genetic ALgorithm system. The aim of Sugal is to support research and implementation in Genetic Algorithms on a common software platform. As such, Sugal supports a large number of variants of Genetic Algorithms, and has extensive features to support customization and extension.

G.2 EC Software Kits/Applications

These are various applications, software kits, etc. meant for research in the field of evolutionary computing. Their ease of use will vary, as they were designed to meet some particular research interest more than as an easy to use commercial package.

ADATE

Web site: www-ia.hiof.no/∼rolando/adate_intro.html

ADATE (Automatic Design of Algorithms through Evolution) is a system for automatic programming i.e., inductive inference of algorithms, which may be the best way to develop artificial and general intelligence.

The ADATE system can automatically generate non-trivial and novel algorithms. Algorithms are generated through large scale combinatorial search that employs sophisticated program transformations and heuristics. The ADATE system is particularly good at synthesizing symbolic, functional programs and has several unique qualities.

esep & xesep

Web site(esep): www.iit.edu/∼elrad/esep.html
Web site(xesep): www.iit.edu/∼elrad/xesep.html

This is a new scheduler, called Evolution Scheduler, based on Genetic Algorithms and Evolutionary Programming. It lives with original Linux priority scheduler. This means the user don't have to reboot to change the scheduling policy. The user may simply use the manager program esep to switch between them at any time, and esep itself is an all-in-one for scheduling status, commands, and administration. We didn't intend to remove the original priority scheduler; instead, at least, esep provides the user with another choice to use a more intelligent scheduler, which carries out natural competition in an easy and effective way.

Xesep is a graphical user interface to the esep (Evolution Scheduling and Evolving Processes). It's intended to show users how to start, play, and feel the Evolution Scheduling and Evolving Processes, including sub-programs to display system status, evolving process status, queue status, and evolution scheduling status periodically in as small as one mini-second.

Corewars

Web site: corewars.sourceforge.net/
SourceForge site: sourceforge.net/projects/corewars/

Corewars is a game which simulates a virtual machine with a number of programs. Each program tries to crash the others. The program that lasts the longest time wins. A number of sample programs are provided and new programs can be written by the player. Screenshots are available at the Corewars homepage.

Grany-3

Web site: zarb.org/∼gc/html/grany.html

Grany-3 is a full-featured cellular automaton simulator, made in C++ with Gtk--, flex++/bison++, doxygen and gettext, useful to granular media physicists.

JCASim

Web site: www.jweimar.de/jcasim/

JCASim is a general-purpose system for simulating cellular automata in Java. It includes a stand-alone application and an applet for web presentations. The cellular automata can be specified in Java, in CDL, or using an interactive dialogue. The system supports many different lattice geometries (1-D, 2-D square, hexagonal, triangular, 3-D), neighborhoods, boundary conditions, and can display the cells using colors, text, or icons.

JGProg

Web site: jgprog.sourceforge.net

Genetic Programming (JGProg) is an open-source Java implementation of a strongly-typed Genetic Programming experimentation platform. Two example "worlds" are provided, in which a population evolves and solves the problem.

Bibliography

1. Goldberg, D. E. and Deb, K.(1991). : A comparative analysis of selection schemes used in genetic algorithms.
2. Luke, S. and Spector, L. (1997). : A Comparison of Crossover and Mutation in Genetic Programming. GP 97- Proceedings of the 2nd Annual Conference, July 13–16, pp. 240–248.
3. J.D. Lohn, G.L. Haith, S.P. Colombano, D. Stassinopoulos, : A Comparison of Dynamic Fitness Schedules for Evolutionary Design of Amplifiers, Proceedings of the First NASA/DoD Workshop on Evolvable Hardware, Pasadena, CA, IEEE Computer Society Press, pages 87–92, 1999.
4. Knowles J.D., Corne D.W., : A Comparison of Encodings and Algorithms for Multiobjective Minimum Spanning Tree Problems, University of Reading, UK.
5. Babovic V et al, : A data mining approach to modelling of water supply assets, Urban Water, 4(2002), pp. 401–414.
6. Whitley D. and Starkweather T, GENITOR II: A distributed genetic algorithm, J. Exp. Theoret. Artif. Intell. 2 (1990), 189–214.
7. B. Paechter, T. Back, M. Schoenauer, M. Sebag, A. E. Eiben, J. J. Merelo, and T. C. Fogarty. : A distributed resource evolutionary algorithm machine (DREAM). In Proceedings of the Congress on Evolutionary Computation 2000 (CEC2000), pages 951–958. IEEE, IEEE Press, 2000.
8. Liang, S., Fuhrman, S., and Somogyi, R. (1998). REVEAL: A general reverse engineering algorithm for inference of genetic network architecture. Proc. Pacific Symposium on Biocomputing 98 pp. 18–29.
9. Edwin S.H. Hou, Nirwan Ansari and Hong Ren (1994), : A Genetic Algorithm for Multiprocessor Scheduling, IEEE Transactions on Parallel and Distributed Systems, Vol. 5, No. 2, pp. 113–120.
10. Glen, R.C. and A.W.R. Payne. : A genetic algorithm for the automated generation of molecules within constraints. Journal of Computer-Aided Molecular Design, vol. 9, p. 181–202 (1995).
11. Soule, Terrence and Amy Ball. : A genetic algorithm with multiple reading frames. In GECCO-2001: Proceedings of the Genetic and Evolutionary Computation Conference, Lee Spector and Eric Goodman (eds). Morgan Kaufmann, 2001. Available online at http://www.cs.uidaho.edu/∼tsoule/research/papers.html.
12. K. Cranmer, R. S. Bowman, PhysicsGP: A genetic programming approach to event selection, Comput. Phys. Commun. 167 (2005) 165.
13. Ptashne, M. (1992). : A Genetic Switch: Phage λ and Higher Organisms. Second Edition. Cambridge, MA: Cell Press and Blackwell Scientific Publications.
14. Smith, S.: A Learning System Based on Genetic Adaptive Algorithms. PhD thesis, Department of Computer Science, University of Pittsburgh (1980).

15. P. Spiessens and B. Manderick, : A massively parallel genetic algorithm, in Proceedings of the Fourth International Conference on Genetic Algorithms (L. B. Booker and R. K. Belew, Eds.), pp. 279–286, Morgan Kaufmann, Los Altos, CA, 1991.
16. Foster and C. Kesselmann, Globus : A metacomputing infrastructure toolkit, J. Supercomput. Appl.11 (1997), 115–128., Los Altos, CA, 1991.
17. Field, P., : A Multary Theory for Genetic Algorithms: Unifying Binary and Non-binary Problem Representations, Ph.D. Thesis, 1995, Queen Mary and Westfield College.
18. Burke, E.K. and J.P. Newall. : A multistage evolutionary algorithm for the timetable problem. IEEE Transactions on Evolutionary Computation, vol. 3, no. 1, p. 63–74 (April 1999).
19. J.R. Rabunal, J. Dorado, A. Pazos, J. Pereira and D. Rivero, : A NewApproach to the Extraction of ANN Rules and to Their Generalization Capacity Through GP. Neural Computation, vol. 16, n. 7. 2004. pp. 1483–1523.
20. Au, Wai-Ho, Keith Chan, and Xin Yao. : A novel evolutionary data mining algorithm with applications to churn prediction. IEEE Transactions on Evolutionary Computation, vol. 7, no. 6, p. 532–545 (December 2003).
21. Abramson D. and Abela J. (1992), : A parallel genetic algorithm for solving the school timetabling problem, in Proceedings of the Fifteenth Australian Computer Science Conference, Australia, Vol. 14, pp. 1–11.
22. Raidl G. R., Drexel C., : A Predecessor Coding in an Evolutionary Algorithm for the Capacitated Minimum Spanning Tree Problem, in Late-Breaking-Papers Proc. of the 2000 Genetic and Evolutionary Computation Conference, Las Vegas, NV, July 2000, 309–316.
23. Fama, E.F., Efficient capital markets : A review of theory and empirical work, Journal of Finance 23, (1970), 383–417.
24. Wang L., Anthony A. Maciejewski, Howard Jay Siegel and Vwani P. Roy Choudhury (2005), : A Study of five parallel Genetic Algorithms using the Traveling Salesman Problem, Intelligent Automation and Soft Computing, Vol. 11, No.4, pp. 214–234.
25. Syswerda G.: A Study of Reproduction in Generational and Steady-State Genetic Algorithms. In Rawlins G. (ed.): Foundations of GAs, Morgan Kaufmann (1991) 94–101
26. Sinclair, M.C., Evolutionary Telecommunications: A Summary, Proc. GECCO99 Workshop on Evolutionary Telecommunications: Past, Present and Future, Orlando, Florida, USA, July 1999, 209–212.
27. Back, T., Hoffmeister, F., & Schwefel, H.-P. (1991) : A survey of evolution strategies. Proceedings of the Fourth International Conference on Genetic Algorithms, 2–9. La Jolla, CA: Morgan Kaufmann.
28. E. Alba and J. M. Troya, : A survey of parallel distributed genetic algorithms, Complexity 4 (1999), 31–52.
29. Cantu-Paz E. (1998), : A survey of parallel genetic algorithms, Calculateurs Paralleles, Reseaux et Systems Repartis, Vol. 10, No. 2, pp. 141–171.
30. Grefenstette, John J. (1989) : A system for learning control strategies with genetic algorithms. Proceedings of the Third International Conference on Genetic Algorithms, 183–190. Fairfax, VA: Morgan Kaufmann.
31. Arkin, A., Peidong, S., and Ross, J. (1997). : A test case of correlation metric construction of a reaction pathway from measurements. Science 277:1275–1279.
32. Lukac, L.P., Brorsen, B.W. & Irwin, S.H., : A test of futures market disequilibrium using twelve different technical trading systems, Applied Economics, 20, (1988), 623–639.
33. J. Sprave, : A unified model of non-panmictic population structures in evolutionary algorithms, in Congress on Evolutionary Computation (CEC99), pp. 1384–1391, IEEE Press, Piscataway, NJ, 1999.
34. Jelasity M. : A Wave Analysis of the Subset Sum Problem. In Back T. (ed.): Proceedings of the Seventh International Conference on Genetic Algorithms. Morgan Kaufmann, San Francisco, CA (1997) 89–96.
35. web site : ACCRE (Advanced Computing Center for Research & Education, Vanderbilt University), http://www.accre.vanderbilt.edu/accre/.
36. J. H. Holland, : Adaptation in Natural and Artificial Systems, The MIT Press, Cambridge, MA, 2nd ed., 1992.

37. Holland, J.H. (1992) : Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence. Ann Arbor, MI: University of Michigan Press 1975. Second edition. Cambridge, MA: The MIT Press 1992.
38. Lipsitch, M. (1991) : Adaptation on rugged landscapes generated by iterated local interactions of neighboring genes. Proceedings of the Fourth International Conference on Genetic Algorithms, 128–135. La Jolla, CA: Morgan Kaufmann.
39. Davis, L. (1989) : Adapting operator probabilities in genetic algorithms. Proceedings of the Third International Conference on Genetic Algorithms, 60–69. La Jolla, CA: Morgan Kaufmann.
40. Holland, J.H., : Adaption in Natural and Artificial Systems, MIT Press, 1975, (1992 Edition).
41. Spector, Lee, Langdon,William B., OReilly, Una-May, and Angeline, Peter (editors). (1999). : Advances in Genetic Programming 3. Cambridge, MA: The MIT Press.
42. Angeline, P. & Kinnear, K.E., (ed.), : Advances in genetic programming II, MIT Press. 1996.
43. Kinnear, K.E. (ed.), : Advances in genetic programming, MIT Press, 1994.
44. Sasaki, Daisuke, Masashi Morikawa, Shigeru Obayashi and Kazuhiro Nakahashi. : Aerodynamic shape optimization of supersonic wings by adaptive range multiobjective genetic algorithms. In Evolutionary Multi-Criterion Optimization: First International Conference, EMO 2001, Zurich, Switzerland, March 2001: Proceedings, K. Deb, L. Theile, C. Coello, D. Corne and E. Zitler (eds). Lecture Notes in Computer Science, vol. 1993, p. 639–652. Springer-Verlag, 2001.
45. Patch, Kimberly. : Algorithm evolves more efficient engine. Technology Research News, June/July 2000. Available online at http://www.trnmag.com/Stories/062800/Genetically_Enhanced_Engine_062800.html.
46. Porto, Vincent, David Fogel and Lawrence Fogel. : Alternative neural network training methods. IEEE Expert, vol. 10, no. 3, p. 16–22 (June 1995).
47. Schaffer, J. D. & Morishima, A. (1987) : An adaptive crossover distribution mechanisms for genetic algorithms. Proceedings of the Second International Conference on Genetic Algorithms, 36–40. Cambridge, MA: Lawrence Erlbaum.
48. Back, T. (1995). Evolution strategies: An alternative evolutionary algorithm. In [Alliot et al., 1995], pages 3–20.
49. Fogel, D. B. (1992) : An analysis of evolutionary programming. Proceedings of the First Annual Conference on Evolutionary Programming, 43–51. La Jolla, CA: Evolutionary Programming Society.
50. Raidl G. R., Julstrom B. A., Edge-sets: An effective evolutionary coding of spanning trees. IEEE Transactions on Evolutionary Computation, 7(3): 2003, 225–239.
51. Zhou G., Gen M., : An effective genetic algorithm approach to the quadratic minimum spanning tree problem, F. J. Computers Ops Res., Vol. 25 (3), 1998, 229–237.
52. Raidl G.R., : An efficient evolutionary algorithm for the degree-constrained minimum spanning tree problem. In C. Fonseca et al., editors, Proceedings of the 2000 IEEE Congress on Evolutionary Computation, IEEE Press, 2000, 104–111.
53. Kobayashi S., Ono I. and Yamamura M. (1995), : An Efficient Genetic Algorithm for Job Shop Scheduling Problems, Proc. of ICGA95, pp. 506–511.
54. M. Munetomo, Y. Takai, and Y. Sato, : An efficient migration scheme for Subpopulation-based ansynchronously parallel genetic algorithms, in Proceedings of the Fifth International Conference on Genetic Algorithms (S. Forrest, Ed.), pp. 649, Morgan Kaufmann Publishers, San Mateo, CA, 1993.
55. E. Cantú-Paz and C. Kamath, : An Empirical Comparison of Combinations of Evolutionary Algorithms and Neural Networks for Classification Problems, IEEE Transactions on systems, Man and Cybernetics – Part B: Cybernetics, 2005, pp. 915–927.
56. Hancock, P. J. (1994). : An empirical comparison of selection methods in evolutionary algorithms.
57. Shumeet Baluja, : An Empirical Comparison of Seven Iterative and Evolutionary Function Optimization Heuristics, Carnegie Mellon University technical report CMU-CS-95–193, September 1995.

58. Reisig, W.: An Informal Introduction To Petri Nets. Proc. Intl Conf. Application and Theory of Petri Nets, Aarhus, Denmark (2000)
59. Mitchell, M., : An Introduction to Genetic Algorithms (Complex Adaptive Systems), MIT Press, 1996
60. Krose, B., van der Smagt, P.: An introduction to neural networks. University of Amsterdam (1996)
61. Duran, M.A. and I.E. Grossmann (1986). : An Outer Approximation Algorithm for a Class of Mixed Integer Nonlinear Programs, Mathematical Programming 36, 307–339.
62. Fonseca, Carlos and Peter Fleming. : An overview of evolutionary algorithms in multiobjective optimization. Evolutionary Computation, vol. 3, no. 1, p. 1–16 (1995).
63. Whitley, D., : An overview of evolutionary algorithms: practical issues and common pitfalls, Information and Software Technology, Vol. 43., 2001, pp. 817–831.
64. Fogel, D. B. (1997). : An overview of evolutionary programming.
65. Coello, Carlos. : An updated survey of GA-based multiobjective optimization techniques. ACM Computing Surveys, vol. 32, no. 2, p. 109–143 (June 2000).
66. Chellapilla, Kumar and David Fogel. : Anaconda defeats Hoyle 6-0: a case study competing an evolved checkers program against commercially available software. In Proceedings of the 2000 Congress on Evolutionary Computation, p. 857–863. IEEE Press, 2000. Available online at http://www.natural-selection.com/NSIPublicationsOnline.htm.
67. P. L. Frabetti, et al., : Analysis of the decay mode $D^+ \to K + \pi + \pi -$, Phys. Lett. B364 (1995) 127–136.
68. W. E. Hart, S. Baden, R. K. Belew, and S. Kohn, : Analysis of the numerical effects of parallelism on a parallel genetic algorithm, in IEEE Proceedings of the Workshop on Solving Combinatorial Optimization Problems in Parallel (IEEE, Ed.), pp. CD-ROM IPPS97, IEEE Press, New York, 1997.
69. E. Alba and J. M. Troya, : Analyzing synchronous and asynchronous parallel distributed genetic algorithms, Future Generation Comput. Systems 17 (2001), 451–465.
70. T. Lai and S. Sahni, : Anomalies in parallel branch-and-bound algorithms, Comm. ACM 27 (1984), 594–602.
71. M. Bot, : Application of Genetic Programming to Induction of Linear Classification Trees, Final Term Project Report, Vrije Universiteit, Amsterdam, 1999.
72. Sato, S., K. Otori, A. Takizawa, H. Sakai, Y. Ando and H. Kawamura. : Applying genetic algorithms to the optimum design of a concert hall. Journal of Sound and Vibration, vol. 258, no. 3, p. 517–526 (2002).
73. T. G. Dietterich, : Approximate statistical tests for comparing supervised classification learning algorithms, Neural Computation, Vol. 10, No. 7, 1998, pp. 1895–1924.
74. De Jong, K. A. (1992) : Are genetic algorithms function optimizers? Proceedings of the Second International Conference on Parallel Problem Solving from Nature.
75. Fogel, L.J., Owens, A. & Walsh, M., : Artificial Intelligence Through Simulated Evolution. New York: John Wiley & Sons, 1966.
76. J. R. Rabunal and J. Dorado, (eds.) : Artificial Neural Networks in Real-Life Applications, Idea Group Inc, 2005.
77. Petzinger, Thomas. : At Deere they know a mad scientist may be a firms biggest asset. The Wall Street Journal, July 14, 1995, p. B1.
78. Yang Y. and Soh C.K, : Automated optimum design of structures using genetic programming, Computers and Structures, Pergamon, 80 (2002), pp. 1537–1546. http://web.mit.edu/emech/dontindex-build/Java/trussworks/index.html
79. Koza, J.R., Keane, M.A., Yu, J., Bennett III, F.H., and Mydlowec,W. (2000a). : Automatic creation of human-competitive programs and controllers by means of genetic programming. Genetic Programming and Evolvable Machines 1:121–164.
80. Koza, J.R., Keane, M.A., Yu, J., Bennett III, F.H., Mydlowec, W., and Stiffelman, O. (1999c). : Automatic synthesis of both the topology and parameters for a robust controller for a non-minimal phase plant and a three-lag plant by means of genetic programming. Proceedings of 1999 IEEE Conference on Decision and Control pp. 5292–5300.

81. Williams, Edwin, William Crossley and Thomas Lang. : Average and maximum revisit time trade studies for satellite constellations using a multiobjective genetic algorithm. Journal of the Astronautical Sciences, vol. 49, no. 3, p. 385–400 (July-September 2001).
82. Naik, Gautam. : Back to Darwin: In sunlight and cells, science seeks answers to high-tech puzzles. The Wall Street Journal, January 16, 1996, p. A1.
83. Lippman, S.B., : C# Primer, A Practical Approach, Addison-Wesley, 2002, (1st Edition).
84. UEA CALMA Group, : Calma Project Report 2.4:Parallelism in Combinatorial Optimisation Technical Report, School of Information Systems, University of East Anglia, Norwich, U.K., September 18, 1995.
85. Whitley D.: Cellular Genetic Algorithms. In Forrest S. (ed.): Proceedings of the Fifth International Conference on GAs. Morgan Kaufmann, San Mateo, CA (1993) 658
86. J. M. Link, et al., : Cerenkov particle identification in FOCUS, Nucl. Instrum. Meth. A484 (2002) 270–286.
87. McAdams, H.H. and Shapiro, L. (1995). : Circuit simulation of genetic networks. Science 269:650–656.
88. S. C. Lin, W. F. Punch, and E. D. Goodman, : Coarse-grain parallel genetic algorithms: Categorization and a new approach, in Sixth IEEE SPDP, pp. 28–37, 1994.
89. Bull, L., Fogarty, T.: Co-evolving communicating classifier systems for tracking. Proc. Intl Conf. Neural Networks and Genetic Algoriths (1993)
90. Schwefel, H.-P. (1987) : Collective Phaenomena in Evolutionary Systems, 31st Annu. Meet. Interl Soc. for General System Research, Budapest, 1025–1033.
91. Milner, R.: Communicating and Mobile Systems: the _-calculus. Cambridge University Press (1999)
92. Hoare, C.: Communicating Sequential Processes. International Series in Computer Science. Prentice Hall (1985)
93. Kirshnamoorthy M., Ernst A. T., : Comparison of Algorithms for the Degree Constrained Minimum Spanning Tree, Journal of Heuristic, 7, 2001, 587–611.
94. Voit, E.O. (2000). : Computational Analysis of Biochemical Systems. Cambridge: Cambridge University Press.
95. Kewley, Robert and Mark Embrechts. : Computational military tactical planning system. IEEE Transactions on Systems, Man and Cybernetics, Part C - Applications and Reviews, vol. 32, no. 2, p. 161–171 (May 2002).
96. Garey M.R., Johnson D.S. : Computers and Intractability, A Guide to the Theory of NPCompleteness .W. H. Freeman, San Francisco, 1979.
97. Lea, D.: Concurrent Programming in Java (2nd edition) Design Principles and Patterns. The Java Series. Addison Wesley (2000)
98. A. Zomorodian, 1995. : Context-free Language Induction by Evolution of Deterministic Push-down Automata Using Genetic Programming, in Working Notes of the Genetic Programming Symposium, AAAI-95, Eric Siegel and John Koza, chairs. AAAI Press. 1995.
99. Assion, A., T. Baumert, M. Bergt, T. Brixner, B. Kiefer, V. Seyfried, M. Strehle and G. Gerber. : Control of chemical reactions by feedback-optimized phase-shaped femtosecond laser pulses. Science, vol. 282, p. 919–922 (30 October 1998).
100. Davidson, Clive. : Creatures from primordial silicon. New Scientist, vol. 156, no. 2108, p. 30–35 (November 15, 1997). Available online at http://www.newscientist.com/hottopics/ai/primordial.jsp.
101. Coale, Kristi. : Darwin in a box. Wired News, July 14, 1997. Available online at http://www.wired.com/news/technology/0,1282,5152,00.html.
102. website: Darwinian selection of satellite orbits for military use. Space.com, 16 October 2001. Available online at http://www.space.com/news/darwin_satellites_011016.html.
103. Mehlhorn K., : Data Structures and algorithms, Vol. II, Graph Algorithms and NPCompleteness, Springer Verlag 1984.
104. Grefenstette, J. G. (1992) : Deception considered harmful. Proceedings of the Foundations of Genetic Algorithms Workshop. Vail, CO: Morgan Kaufmann.
105. P. L. Frabetti, et al., : Description and performance of the Fermilab E687 spectrometer, Nucl. Instrum. Meth. A320 (1992) 519–547.

106. Altshuler, Edward and Derek Linden. : Design of a wire antenna using a genetic algorithm. Journal of Electronic Defense, vol. 20, no. 7, p. 50–52 (July 1997).
107. Koppen M and Nickolay B, : Design of image exploring agent using genetic programming, Proceedings of IIZUKA96 Japan, 1996, pp. 549–552.
108. Lee, Yonggon and Stanislaw H. Zak. : Designing a genetic neural fuzzy antilock-brake-system controller. IEEE Transactions on Evolutionary Computation, vol. 6, no. 2, p. 198–211 (April 2002).
109. Giro, R., M. Cyrillo and D.S. Galvão. : Designing conducting polymers using genetic algorithms. Chemical Physics Letters, vol. 366, no. 1–2, p. 170–175 (November 25, 2002).
110. Mittenthal, J.E., Ao, Y., Bertrand C., and Scheeline, A. (1998). : Designing metabolism: Alternative connectivities for the pentose phosphate pathway. Bulletin of Mathematical Biology 60:815–856.
111. H. Kitano, : Designing neural networks using genetic algorithms with graph generation system, Complex Systems, vol. 4, 1990, pp. 461–476.
112. Roberts S.C. and Howard D, : Detection of incidents on motorways in low flow high speed conditions by genetic programming, Cagnoni S et al (eds): EvoWorkshops 2002, LNCS 2279, Springer-Verlag, 2002, pp. 245–254.
113. J. L. Gustafson, G. R. Montry, and R. E. Brenner, : Development of parallel methods for a 1024- processor hypercube, SIAM Sci. Statist. Comput. 9 (1988), 609–638.
114. Isermann, R. (1989). : Digital Control Systems, Vol. 1, 2nd Edition, Springer-Verlag. Koza, J. (1992). Genetic programming: On the programming of computers by means of natural selection, The MIT press.
115. R. Tanese, : Distributed genetic algorithms in Proceedings of the Third International Conference on Genetic Algorithms (J. D. Schaffer, Ed.), pp. 434–439, Morgan kaufmann, Los Altos, CA, 1989.
116. Goldberg, D. E., Deb, K., & Korb, B. (1991) : Dont worry, be messy. Proceedings of the Fourth International Conference on Genetic Algorithms, 24–30. La Jolla, CA: Morgan Kaufmann.
117. Schraudolph, N. N., & Belew, R. K. (1992) : Dynamic parameter encoding for genetic algorithms. Machine Learning Journal, Volume 9, Number 1, 9–22.
118. Chryssolouris, George and Velusamy Subramaniam. : Dynamic scheduling of manufacturing job shops using genetic algorithms. Journal of Intelligent Manufacturing, vol. 12, no. 3, p. 281–293 (June 2001).
119. Sambridge, Malcolm and Kerry Gallagher. : Earthquake hypocenter location using genetic algorithms. Bulletin of the Seismological Society of America, vol. 83, no. 5, p. 1467–1491 (October 1993).
120. E. Lutton, P. Collet, and J. Louchet. : Easea comparisons on test functions : Galib versus eo. In EA01: 5th conference on Artificial Evolution. Le Creusot, September 2001.
121. Tomita, M., Hashimoto, K., Takahashi, K., Shimizu, T.S., Matsuzaki, Y., Miyoshi, F., Saito, K., Tanida, S., Yugi, K., Venter, J.C., Hutchison, C.A. (1999). : E-CELL: Software environment for whole cell simulation. Bioinformatics 15(1):72–84.
122. Tsang, E.P.K., Li, J. & Butler, J.M., : EDDIE beats the bookies, International Journal of Software, Practice & Experience, Wiley, Vol.28 (10), 1033–1043, August 1998.
123. Butler, J.M., : Eddie beats the market, data mining and decision support through genetic programming, Developments, Reuters Limited, (1997), Vol1.
124. Malkiel, B., : Efficient market Hypothesis, in Newman,P., Milgate, M. and Eatwell, J.(eds.), New Palgrave Dictionary of Money and Finance, Macmillan, London, (1992), pp. 739.
125. Graham-Rowe, Duncan. : Electronic circuit evolves from liquid crystals. New Scientist, vol. 181, no. 2440, p. 21 (March 27, 2004).
126. Ashour A.F et al, : Empirical modelling of shear strength of RC deep beams by genetic programming, Computers and Structures, Pergamon, 81 (2003), pp. 331–338.
127. Ashley, Steven. : Engineous explores the design space. Mechanical Engineering, February 1992, p. 49–52.
128. Ishino Y and Jin Y, : Estimate design intent: a multiple genetic programming and multivariate analysis based approach, Advanced Engineering Infomatics, 16(2002), pp. 107–125.

129. Jegadeesh, N, : Evidence of predictable behaviour of security returns, Journal of Finance 45, No 3, (1990), 881–898.
130. Maynard-Smith, J. (1982). : Evolution and the Theory of Games. Cambridge University Press.
131. Rao, Srikumar. : Evolution at warp speed. Forbes, vol. 161, no. 1, p. 82–83 (January 12, 1998).
132. S. Nolfi and D. Parisi, : Evolution of Artificial Neural Networks, Handbook of brain theory and neural networks, Second Edition, Cambridge, MA: MIT Press, 2002, pp. 418–421.
133. R. Poli : Evolution of Graph-like Programs with Parallel Distributed Genetic Programming, Genetic Algorithms: Proceedings of the Seventh International Conference, 1997.
134. Koza, J.R. : Evolution of subsumption using genetic programming. In Varela, F.J., Bourgine, P., eds.: Proceedings of the First European Conference on Arti_cial Life. Towards a Practice of Autonomous Systems, Paris, France, MIT Press (1992) 110–119
135. Kursawe, F. (1992) : Evolution strategies for vector optimization, Taipei, National Chiao Tung University, 187–193.
136. Kursawe, F. (1994) : Evolution strategies: Simple models of natural processes?, Revue Internationale de Systëmique, France (to appear).
137. Mesghouni K., Slim Hammadi and Pierre Borne (2004), : Evolutionary algorithms for Job shop scheduling, Int. Journal of Appl. Math. Comput. Sci., Vol.14, No.1, pp. 91–103.
138. Fleming, P.J., Purshouse, R.C., : Evolutionary algorithms in control systems engineering: a survey, Control Engineering Practice, Vol. 10., 2002, pp. 1223–1241.
139. Dasgupta, D., Michalewicz, Z., : Evolutionary Algorithms in Engineering Applications, Springer Verlag, 1997
140. T. Back, : Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms, Oxford Univ. Press, New York, 1996.
141. In Davis, L. D., De Jong, K., Vose, M. D., and Whitley, L. D., editors, : Evolutionary Algorithms, pages 89–109. Springer.
142. In Fogarty, T. C., editor, : Evolutionary Computing (AISB94), pages 80–94.
143. Dracopoulos, D.C. (1997). : Evolutionary Control of a Satellite, GP 97- Proceedings of the 2nd Annual Conference, July 13–16, pp. 77–81.
144. Hofbauer, J. and Sigmund,K. (1998). : Evolutionary Games and Population Dynamics. Cambridge University Press.
145. Romaniuk, S. G.: Evolutionary Growth Perceptrons. In Forrest S. (ed.): Proceedings of the Fifth International Conference on Genetic Algorithms. Morgan Kaufmann, San Mateo, CA (1993) 334–341.
146. Box, G. E. P. (1957) : Evolutionary operation: a method of increasing industrial productivity. Applied Statistics, Vol. 6, 81–101.
147. Hong YS and Bhamidimarri R, : Evolutionary self-organising modelling of a municipal wastewater treatment plant, Water Research, 37(2003), pp. 1199–1212.
148. Fogel, D. B. and Fogel, G. B.(1995). : Evolutionary stable strategies are not always stable under evolutionary dynamics. In Evolutionary Programming IV, pages 565–577.
149. Weibull, J. (1995). : EvolutionaryGame Theory, MIT Press.
150. Samuelson, L. (1997). : EvolutionaryGames and Equilibrium Selection. MIT Press.
151. Rechenberg, I. (1973) : Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution, Stuttgart: Fromman-Holzboog.
152. Koza, J. R. (1991) : Evolving a computer program to generate random numbers using the genetic programming paradigm. Proceedings of the Fourth International Conference on Genetic Algorithms, 37–44. La Jolla, CA: Morgan Kaufmann.
153. Chellapilla, Kumar and David Fogel. : Evolving an expert checkers playing program without using human expertise. IEEE Transactions on Evolutionary Computation, vol. 5, no. 4, p. 422–428 (August 2001). Available online at http://www.natural-selection.com/NSIPublicationsOnline.htm.
154. Montana D.J. and Czerwinski S, : Evolving control laws for a network of traffic signals, Proceedings of the First Annual Conference: Genetic Programming, July 28-3, 1996. Stanford University, pp. 333–338.

155. S. Luke and L. Spector, : Evolving Graphs and Networks with Edge encoding: Preliminary Report. In Late Breaking Papers at the Genetic Programming 1996 Conference (GP96). J. Koza, ed. Stanford: Stanford Bookstore, 1996, pp. 117–124.
156. Koza, John, Martin Keane and Matthew Streeter. : Evolving inventions. Scientific American, February 2003, p. 52–59.
157. M. Keijzer, J. J. Merelo, G. Romero, and M. Schoenauer. : Evolving objects:a general purpose evolutionary computation library. In EA01: 5th conference on Artificial Evolution. Le Creusot, September 2001.
158. Rizki, Mateen, Michael Zmuda and Louis Tamburino. : Evolving pattern recognition systems. IEEE Transactions on Evolutionary Computation, vol. 6, no. 6, p. 594–609 (December 2002).
159. Eisenstein, J. : Evolving robocode tank _ghters. Technical Report AIM-2003-023, AI Lab, Massachusetts Institute Of Technology (2003) citeseer.ist.psu.edu/647963.html.
160. Sharman, K., Esparcia-Alcazar, A. I. and Li, Y. (1995). : Evolving Signal Processing Algorithms using Genetic Programming, IEE conference on GAs in Engineering Systems, Conf. Number 414, September12–14.
161. Andre, David and Astro Teller. : Evolving team Darwin United. In RoboCup-98: Robot Soccer World Cup II, Minoru Asada and Hiroaki Kitano (eds). Lecture Notes in Computer Science, vol. 1604, p. 346–352. Springer-Verlag, 1999.
162. Andreou, Andreas, Efstratios Georgopoulos and Spiridon Likothanassis. : Exchange-rates forecasting: A hybrid algorithm based on genetically optimized adaptive neural networks. Computational Economics, vol. 20, no. 3, p. 191–210 (December 2002).
163. Yuh, C.-H., Bolouri, H., and Davidson, E.H. (1998). Genomic cisregulatory logic: Experimental and computational analysis of a sea urchin gene. Science 279:1896–1902.
164. Z. Fan, K. Seo, R. C. Rosenberg, J. Hu and E. D. Goodman, : Exploring Multiple Design Topologies Using Genetic Programming And Bond Graphs. GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference. Springer-Verlag. 2002, pp. 1073–1080
165. Lehmann, B.N., : Fad, martingales, and market efficiency, Quarterly Journal of Economics, 105, (1990), 1–28.
166. Fama, E.F. & Blume, M.E., : Filter rules and stock-market trading, Journal of Business 39(1), (1966), 226–241.
167. Mahfoud, S. & Mani, G., : Financial Forecasting Using Genetic Algorithms, Journal of Applied Artificial Intelligence Vol. 10, Num 6, (1996), 543–565.
168. Mahfoud, Sam and Ganesh Mani. : Financial forecasting using genetic algorithms. Applied Artificial Intelligence, vol. 10, no. 6, p. 543–565 (1996).
169. Ovalle-Martinez F.J, Stojmenovic I, Garcia-Nocetti F., Solano-Gonzalez J.,: Finding minimum transmission radii for preserving connectivity and constructing minimal spanning tree in ad hoc and sensor networks, J. Parallel Distrib. Comput., 65, 2005, 132–141.
170. Smith, S. (1983) : Flexible learning of problem solving heuristics through adaptive search. Proceedings of the Eighth International Joint Conference on Artificial Intelligence, 422–425. Karlsruche, Germany: William Kaufmann.
171. In Rawlins, G. J., editor, : Foundations f Genetic Algorithms (FOGA 1), pages 69–93.
172. Tsang, E.P.K. : Foundations of constraint satisfaction, Academic Press, London, 1993. Improving Technical Analysis Predictions Page 13 of 13
173. E. Alba, J. F. Aldana and J. M. Troya, : Fully automatic ANN design: A genetic approach, Proc. Int. Workshop Artificial Neural Networks (IWANN93), Lecture Notes in Computer Science, vol. 686. Berlin, Germany: Springer-Verlag, 1993, pp. 399–404.
174. Werner, F.M., Bondt, D. & Thaler, R., : Further Evidence on Investor Overreaction and Stock Market Seasonality, Journal of Finance, 42, no. 3, (July 1987), 557–581.
175. Dugatkin, L. A. and Reeve,H. K., editors (1998). : Game Theory and Animal Behavior. Oxford University Press.
176. Geoffrion, A.M. (1972). : Generalized Benders Decomposition, Journal of Optimization Theory and Applications 10(4) 237–260.

177. Jensen, Mikkel. : Generating robust and flexible job shop schedules using genetic algorithms. IEEE Transactions on Evolutionary Computation, vol. 7, no. 3, p. 275–288 (June 2003).
178. Kidwell M.D. and Cook D.J. (1994), : Genetic Algorithm for Dynamic Task Scheduling, in Proceedings of the IEEE 13th Annual International Phoenix Conference Computers and Communication, pp. 61–67.
179. Mak B., Blanning R., Ho S., : Genetic algorithm in logic tree decision modeling, European Journal of Operation Research 170, 2006, 597–612.
180. Z. Michalewicz, : Genetic Algorithms+Data Structures=Evolution Programs, 3rd ed., Springer- Verlag, Heidelberg, 1996.
181. Bauer, R. J. Jr., : Genetic Algorithms and Investment Strategies. New York, John Wiley & Sons, Inc., 1994.
182. Tang, K.S., Man, K.F., Kwong, S. and He., Q., : Genetic Algorithms and their applications, IEEE Signal Processing Magazine, November 1996, pp. 22–37.
183. Tang, K.S., K.F. Man, S. Kwong and Q. He. : Genetic algorithms and their applications. IEEE Signal Processing Magazine, vol. 13, no. 6, p. 22–37 (November 1996).
184. Alba E., Aldana J. F., Troya J. M.: Genetic Algorithms as Heuristics for Optimizing ANN Design. In Albrecht R. F., Reeves C. R., Steele N. C. (eds.): Artificial Neural Nets and Genetic Algorithms. Springer-Verlag (1993) 683–690
185. Charbonneau, Paul. : Genetic algorithms in astronomy and astrophysics. The Astrophysical Journal Supplement Series, vol. 101, p. 309–334 (December 1995).
186. Goldberg, D.E., : Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wesley, 1989
187. Miller, B.L. & Goldberg, D.E., : Genetic Algorithms, tournament selection, and the effects of Noise. IlliGAL Report No. 95006. 1995.
188. Holland, John. : Genetic algorithms. Scientific American, July 1992, p. 66–72.
189. Forrest, Stephanie. : Genetic algorithms: principles of natural selection applied to computation. Science, vol. 261, p. 872–878 (1993).
190. Fleurent and Ferland J.A. (1995), : Genetic and hybrid algorithms for graph coloring, Annals of Operations Research, Vol.63, pp. 437–463.
191. F. Gruau, : Genetic micro programming of neural networks, in Kinnear, Jr., K. E., editor, Advances in Genetic Programming, chapter 24, MIT Press, 1994, pp. 495–518.
192. Banzhaf, W., Nordin,P., Keller, R.E, Francone, F.D., : Genetic Programming : An Introduction : On the Automatic Evolution of Computer Programs and Its Applications, Morgan Kaufmann Publishers, ISBN 1-55860-510-X, 1998.
193. Koza, J.R., Banzhaf, W., Chellapilla, K., Deb, K., Dorigo, M., Fogel, D.B., Garzon, M.H., Goldberg, D.E., Iba, H., and Riolo, R. (editors). (1998). : Genetic Programming 1998: Proceedings of the Third Annual Conference. San Francisco, CA: Morgan Kaufmann.
194. Langdon, W.B. (1998). : Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming! Amsterdam: Kluwer. Loomis,W.F. and Sternberg, P.W. (1995). Genetic networks. Science 269:649.
195. Howley, B. (1997). : Genetic Programming and Parametric Sensitivity: a Case Study in Dynamic Control of a Two- Link Manipulator, GP 97- Proceedings of the 2nd Annual Conference, July 13–16, pp. 180–185.
196. Koza, J.R. (1994b). : Genetic Programming II Videotape: The Next Generation. MIT Press.
197. Koza, J.R. (1994a). : Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press.
198. Koza, J.R., Bennett III, F.H, Andre, D., Keane, M.A., and Brave, S. (1999b). : Genetic Programming III Videotape: Human-Competitive Machine Intelligence. San Francisco, CA: Morgan Kaufmann.
199. Koza, John, Forest Bennett, David Andre and Martin Keane. : Genetic Programming III: Darwinian Invention and Problem Solving. Morgan Kaufmann Publishers, 1999.
200. Koza, J.R., Bennett III, F.H, Andre, D, and Keane, M.A. (1999a). : Genetic Programming III: Darwinian Invention and Problem Solving. San Francisco, CA: Morgan Kaufmann.

201. Koza, J.R., Keane, M.A., Streeter, M.J., Mydlowec, W., Yu, J., Lanza, G. : Genetic Programming IV: Routine Human-Competitive Machine Intelligence. Kluwer Academic Publishers, Norwell, MA (2003)
202. Lee D.G et al, : Genetic programming model for long-term forecasting of electric power demand, Electric power systems research, Elsevier, 40, 1997, pp. 17–22.
203. Howley, B. (1996). : Genetic Programming of Near- Minimum-Time Spacecraft Attitude Manoeuvres, GP 96- Proceedings of the 1st Annual Conference, July 28–31, pp. 98–106.
204. Poli, R., Banzhaf, W., Langdon, W.B., Miller, J., Nordin, P., and Fogarty, T.C. (2000). : Genetic Programming: European Conference, EuroGP 2000, Edinburgh Scotland, UK, April 2000, Proceedings. Lecture Notes in Computer Science. Volume 1802. Berlin, Germany: Springer-Verlag.
205. de Garis, H. (1990) : Genetic programming: modular evolution for darwin machines. Proceedings of the 1990 International Joint Conference on Neural Networks, 194–197. Washington, DC: Lawrence Erlbaum.
206. J. R. Koza, : Genetic Programming: On the Programming of Computerby Means of Natural Selection, Cambridge, MA, MIT Press, 1992.
207. Koza, J.R., : Genetic Programming: on the programming of computers by means of natural selection. MIT Press, 1992.
208. Koza, J.R. (1992). : Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press.
209. Sette, S., Boullart, L., : Genetic programming: principles and applications, Engineering Application of Artificial Intelligence, Vol. 14., 2001, pp. 727–736.
210. Koza, J.R.: Genetic Programming; on the programming of computers by means of natural selection. MIT Press (1992)
211. Easton, R.W. (1998). : Geometric methods fordiscrete dynamical systems. Oxford University Press.
212. Hanne, Thomas. : Global multiobjective optimization using evolutionary algorithms. Journal of Heuristics, vol. 6, no. 3, p. 347–360 (August 2000).
213. Raidl G.R., Julstrom B. A., : Greedy Heuristics and an Evolutionary Algorithm for the Bounded-Diameter Minimum Spanning Tree Problem, in Proc. of the 2003 ACM Symposium on Applied Computing, March 2000, 747–752.Rechenberg, I. (1973) Evolutionsstrategie: Optimierung Technischer Systeme nach Prinzipien der Biologischen Evolution. Frommann-Holzboog, Stuttgart.
214. Back T., Fogel D., Michalewicz Z. (eds.) : Handbook of Evolutionary Computation. Oxford University Press (1997)
215. Alba E., Nebro A.J. and Troya J.M. (2002), : Heterogeneous computing and parallel genetic algorithms, Journal of Parallel and Distributed Computing, Vol. 62, No.9, pp. 1362–1385.
216. Grefenstette, J. G., and Baker, J. E. (1989) : How genetic algorithms work: a critical look at implicit parallelism. Proceedings of the Third International Conference on Genetic Algorithms, 20–27. Fairfax, VA: Morgan Kaufmann.
217. Sterling, T.L., Salmon, J., and Becker, D.J., and Savarese, D.F. (1999). : How to Build a Beowulf: A Guide to Implementation and Application of PC Clusters. Cambridge, MA: MIT Press.
218. Galinier P. and Hao J.K. (1999), : Hybrid evolutionary algorithm for graph coloring, Journal of Combinatorial Optimization, Vol.3, No.4, pp. 373–397.
219. He, L. and N. Mort. : Hybrid genetic algorithms for telecommunications network back-up routeing. BT Technology Journal, vol. 18, no. 4, p. 42–50 (Oct 2000).
220. Kojima F. et al, : Identification of crack profiles using genetic programming and fuzzy inference, Journal of Materials Processing Technology, Elsevier, 108 (2001), pp. 263–267.
221. Janikow, C. (1991) : Inductive learning of decision rules from attribute-based examples: A knowledge-intensive genetic algorithm approach. TR91–030, The University of North Carolina at Chapel Hill, Dept. of Computer Science, Chapel Hill, NC.
222. Beale, E. M. L. (1977). : Integer Programming, in The State of the Art in Numerical Analysis (D. Jacobs, ed.) Academic Press: London, 409–448.

223. Goonatilake, S. & Treleaven, P. (ed.), : Intelligent systems for finance and business, Wiley, New York, 1995.
224. Teller A. and M. Veloso, : Internal reinforcement in a connectionist genetic programming approach, Artificial Intelligence. Vol. 120, N. 2, 2000, pp. 165–198.
225. Neely, C., Weller, P. & Ditmar, R., : Is technical analysis in the foreign exchange market profitable? A genetic programming approach, in Dunis, C. & Rustem, B.(ed.), Proceedings, Forecasting Financial Markets: Advances for Exchange Rates, Interest Rates and Asset Management, London. 1997.
226. De Jong, K. & Spears, W. (1991) : Learning concept classification rules using genetic algorithms. Proceedings of the Twelfth International Joint Conference on Artificial Intelligence, 651–656. Sydney, Australia: Morgan Kaufmann.
227. Dhaeseleer, P.,Wen, X., Fuhrman, S., and Somogyi, R. (1999). : Linear modeling of mRNA expression levels during CNS development and injury. Proc. Paficic Symposium on Biocomputing99 pp. 41–52.
228. W. Kantschik, W. Banzhaf, : Linear-Graph GP - A new GP Structure, in Proceedings of the 4th European Conference on Genetic Programming, EuroGP 2002, 2002.
229. Lemley, Brad. : Machines that think. Discover, January 2001, p. 75–79.
230. J. M. Link, et al., : Measurements of the q2 dependence of the $D^+ \to K + \pi + \pi +$ and $D^+ \to K + \pi + \pi -$ form factors, Phys. Lett. B607 (2005) 233–242.
231. R. Gillet, : Memory channel network for PCI, IEEE Micro. 16 (1996), 12–18.
232. W. Kantschik, P. Dittrich, M. Brameier and W. Banzhaf, : MetaEvolution in Graph GP, Proceedings of EuroGP99, LNCS, Vol. 1598. SpringerVerlag, 1999, pp. 15–28.
233. lil-gp web site,: Michigan State University GARARGe group, http://garage. cps.msu.edu/software/lil-gp/lilgp-index.html.
234. Sahinidis, N. and I.E. Grossmann (1989). MINLP Model for Scheduling in Continuous Parallel Production Lines. Presented at AIChE meeting, San Francisco, CA.
235. Grossman, I. E. (1990). : Mixed-Integer Nonlinear Programming Techniques for the Synthesis of Engineering Systems, : Research in Engineering Design. 1, 205–228.
236. Whigham P.A. and Crapper P.F, : Modelling rainfall-runoff using genetic programming, Mathematical and Computer Modelling, 33(2001), pp. 707–721.
237. web site: MSU Genetic Algorithms Research and Applications Group (GARAGe), http://garage.cps.msu.edu/software/software-index.html.
238. Zitzler, Eckart and Lothar Thiele. : Multiobjective evolutionary algorithms: a comparative case study and the Strength Pareto approach. IEEE Transactions on Evolutionary Computation, vol. 3, no. 4, p. 257–271 (November 1999).
239. Obayashi, Shigeru, Daisuke Sasaki, Yukihiro Takeguchi, and Naoki Hirose. : Multiobjective evolutionary computation for supersonic wing-shape optimization. IEEE Transactions on Evolutionary Computation, vol. 4, no. 2, p. 182–187 (July 2000).
240. Srinivas, N. and Kalyanmoy Deb. : Multiobjective optimization using nondominated sorting in genetic algorithms. Evolutionary Computation, vol. 2, no. 3, p. 221–248 (Fall 1994).
241. Neftci, S.N., : Naïve trading rules in financial markets and Wiener-Kolmogorov prediction theory: A study of technical analysis, Journal of Business, 64, (1991), 549–571.
242. S. Haykin, : Neural Networks (2nd ed.), Englewood Cliffs, NJ: Prentice Hall, 1999.
243. Cammarata G., Cavalieri S., Fichera A., Marletta L. : Noise Prediction in Urban Traffic by a Neural Approach. In Mira J., Cabestany J., Prieto A. (eds.): Proceedings of the International Workshop on Artificial Neural Networks, Springer-Verlag (1993) 611–619
244. Mendes, P. and Kell, D.B. (1998). : Non-linear optimization of biochemical pathways: Applications to metabolic engineering and parameter estimation. Bioinformatics 14(10):869–883.
245. Schwefel, H.-P. (1981) : Numerical Optimization of Computer Models. New York: John Wiley & Sons.
246. Schwefel, H.-P. (1977) : Numerische Optimierung von Computermodellen mittels der Evolutionsstrategie, Basel: Birkhäuser.
247. Albert Y. Zomaya and Yee-Hwei (2001), : Observations on using Genetic Algorithms for Dynamic Load Balancing, IEEE Transactions on Parallel and Distributed Systems, Vol. 12, No. 9, pp. 899–911.

248. Schaffer, J. D., Eshelman, L. J. (1991) : On crossover as an evolutionarily viable strategy. Proceedings of the Fourth International Conference on Genetic Algorithms, 61–68. La Jolla, CA: Morgan Kaufmann.
249. Haas, O.C.L., K.J. Burnham and J.A. Mills. : On improving physical selectivity in the treatment of cancer: A systems modelling and optimisation approach. Control Engineering Practice, vol. 5, no. 12, p. 1739–1745 (December 1997).
250. Fogel, G. B., Andrews, P. C., and Fogel,D. B. (1998). : On the instability of evolutionary stable strategies in small populations. Ecological Modelling,109:283–294.
251. Xu W.: On the quadratic minimum spanning tree problem. Proceedings of 1995 Japan- China International Workshop on Information Systems, eds. M. Gen and W. Xu, Ashikaga, 1995, 141–148.
252. Benini, Ernesto and Andrea Toffolo. : Optimal design of horizontal-axis wind turbines using blade-element theory and evolutionary computation. Journal of Solar Energy Engineering, vol. 124, no. 4, p. 357–363 (November 2002).
253. Rechenberg, I., Evolutionsstratrategie: Optimierung technischer systeme nach prinzipien der biologischen evolution. Stuttgart: Frommann-Holzboog, 1973.
254. Kirkpatrick, S., C.D. Gelatt and M.P. Vecchi. : Optimization by simulated annealing. Science, vol. 220, p. 671–678 (1983).
255. Bilbro, G.L. and W.E. Snyder (1991). : Optimization of functions with many minima, IEEE Transactions on Systems, Man, and Cybernetics 21(4), 840–849.
256. Shonkwiler R. : Parallel Genetic Algorithms. In Forrest S. (ed.): Proceedings of the Fifth International Conference on Genetic Algorithms. Morgan Kaufmann, San Mateo, CA (1993) 199–205
257. P. Cal!egari, F. Guidec, P. Kuonen, and D. Kobler, : Parallel island-based genetic algorithm for radio network design. J. Parallel Distrib. Comput. 47 (1997), 86–90.
258. Jeremy Frank, Ari Jonsson, Robert Morris, and David Smith, : Planning and Scheduling for Fleets of Earth Observing Satellites, Proceedings of the 6th International Symposium on Artificial Intelligence, Robotics, Automation and Space 2002, Montreal, 18–22 June 2002.
259. Oussaidene, M., Chopard, B., Pictet, O. & Tomassini, M., : Practical aspects and experiences – Parallel genetic programming and its application to trading model induction, Journal of Parallel Computing Vol. 23, No. 8, (1997), 1183–1198.
260. Haupt, Randy and Sue Ellen Haupt. : Practical Genetic Algorithms. John Wiley & Sons, 1998.
261. E. Cantlu-Paz and D. E. Goldberg, : Predicting speedups of idealized bounding cases of parallel genetic algorithms, in Proceedings of the Seventh International Conference on Genetic Algorithms (T. Back, Ed.), pp. 113–120, Morgan Kaufmann, Los Altos, CA, 1997.
262. Dorado J et al, : Prediction and modelling of the flow of a typical urban basin through genetic programming, Cagnoni S et al (eds): EvoWorkshops 2002, LNCS 2279, Springer-Verlag, 2002, pp. 190–201.
263. J. R. Rabunal, J. Dorado, J. Puertas, A. Pazos, A. Santos and D. Rivero, : Prediction and Modelling of the Rainfall-Runoff Transformation of a Typical Urban Basin using ANN and GP, Applied Artificial Intelligence, 2003.
264. Eshelman, L. J., & Schaffer, J. D. (1991) : Preventing premature convergence in genetic algorithms by preventing incest. Proceedings of the Fourth International Conference on Genetic Algorithms, 115–122. La Jolla, CA: Morgan Kaufmann.
265. Alexander, S.S., : Price movement in speculative markets: Trend or random walks, No. 2, in Cootner, P. (ed.), the random character of stock market prices, MIT Press, Cambridge, MA, 1964, 338–372.
266. Koza, J., Goldberg, D., Fogel, D. & Riolo, R. (ed.), : Procedings of the First Annual Confrence on Genetic programming, MIT Press, 1996.
267. Fogel, D. B., & Atmar, J. W. (eds.) (1992) : Proceedings of the First Annual Conference on Evolutionary Programming. La Jolla, CA: Evolutionary Programming Society
268. Belew, R. K., & Booker, L. B. (eds.) (1991) : Proceedings of the Fourth International Conference on Genetic Algorithms. La Jolla, CA: Morgan Kaufmann.

269. Whitley, D., Goldberg, D., Cantu-Paz, E., Spector, L., Parmee, I., and Beyer, H.-G. (editors). (2000). GECCO-2000: Proceedings of the Genetic and Evolutionary Computation Conference, July 10–12, 2000, Las Vegas, Nevada. San Francisco: Morgan Kaufmann Publishers.
270. Manner, R., & Manderick, B. (1992) : Proceedings of the Second International Conference on Parallel Problem Solving from Nature, Amsterdam: North Holland. Levine D. (1994), A Parallel Genetic Algorithm for the Set Partitioning Problem. T. R. No. ANL-94/23, Argonne National Laboratory, Mathematics and Computer Science Division.
271. Back, T., (ed.), : Proceedings of the seventh international conference on genetic algorithms, San Francisco, California: Morgan Kaufmann Publishers, Inc., 1997.
272. Morgan Kaufmann.: Proceedings of the Third International Conference on Genetic Algorithms, 70- 79. Fairfax, VA
273. V. Donaldson, F. Berman, and R. Paturi, : Program speedup in a heterogeneous computing network, J. Parallel Distrib. Comput. 21 (1994), 316–322.
274. L. Wall, T. Christiansen, R. L. Schwartz, : Programming Perl, 2nd Edition, OReilly & Associates, Inc., Sebastopol, California, 1996.
275. Gibbs, W. Wayt. : Programming with primordial ooze. Scientific American, October 1996, p. 48–50.
276. Graham-Rowe, Duncan. : Radio emerges from the electronic soup. New Scientist, vol. 175, no. 2358, p. 19 (August 31, 2002). Available online at http://www.newscientist.com/news/news.jsp?id=ns99992732.
277. Gillet, Valerie. : Reactant- and product-based approaches to the design of combinatorial libraries. Journal of Computer-Aided Molecular Design, vol. 16, p. 371–380 (2002).
278. Gen M., Kumar A., Kim J.R., : Recent network design techniques using evolutionary algorithms, Int. J. of Production Economics, 98, 2006, 251–261.
279. Booker, L. B. (1992) : Recombination distributions for genetic algorithms. Proceedings of the Foundations of Genetic Algorithms Workshop. Vail, CO: Morgan Kaufmann.
280. Sagan, Carl. Brocas Brain: Reflections on the Romance of Science. Ballantine, 1979.
281. Sutton, R.S., Barto, A.G.: Reinforcement Learning – An Introduction. MIT Press (1998)
282. Rothlauf F,: Representations for Genetic and Evolutionary Algorithms, Studies in Fuzziness and Soft Computing. Physica, Heidelberg, 2002.
283. Palmer C.C., Kershenbaum A., : Representing trees in genetic algorithms, in Proceedings of the First IEEE Conference on Evolutionary Computation, David Scharrer, Hans-Paul Schwefel, and David B. Fogel, Eds., IEEE Press, 1994, 379–384.
284. Jegadeesh, N. & Titman, S., : Returns to Buying Winners and Selling Losers: Implications for Stock Market Efficiency, Journal of Finance, 48, No.1, (1993), 65–91.
285. Koza, J.R., Mydlowec, W., Lanza, G., Yu, J., and Keane, M.A. (2000b). : Reverse Engineering and Automatic Synthesis of Metabolic Pathways from Observed Data Using Genetic Programming. Stanford Medical Informatics Technical Report SMI-2000–0851.
286. Adamidis P. (1994), : Review of Parallel Genetic Algorithms Bibliography, Technical report, Automation and Robotics Lab., Dept. of Electrical and Computer Eng., Aristotle Univ. of Thessaloniki, Greece.
287. K. Hagiwara, et al., : Review of Particle Physics, Phys. Rev. D 66 (2002) 010001. [12] J. M. Link, et al., Study of the doubly and singly Cabibbo suppressed decays $D+ \to K+\pi+\pi+$ and $D+ \to K+\pi+\pi-$, Phys. Lett. B601 (2004) 10–19.
288. Weismann, Dirk, Ulrich Hammel, and Thomas Bäck. : Robust design of multilayer optical coatings by means of evolutionary algorithms. IEEE Transactions on Evolutionary Computation, vol. 2, no. 4, p. 162–167 (November 1998).
289. Koza, John, Martin Keane, Matthew Streeter, William Mydlowec, Jessen Yu and Guido Lanza. Genetic Programming IV : Routine Human-Competitive Machine Intelligence. Kluwer Academic Publishers, 2003.
290. Beasley, J.E., J. Sonander and P. Havelock. : Scheduling aircraft landings at London Heathrow using a population heuristic. : Journal of the Operational Research Society, vol. 52, no. 5, p. 483–493 (May 2001).

291. Al Globus, James Crawford, Jason Lohn, and Robert Morris, : Scheduling Earth Observing Fleets Using Evolutionary Algorithms: Problem Description and Approach, Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space, NASA, Houston,Texas, 27–29 October, 2002.
292. Gordon V. S., Whitley D. : Serial and Parallel Genetic Algorithms as Function Optimizers. In Forrest S. (ed.): Proceedings of the Fifth International Conference on Genetic Algorithms. Morgan Kaufmann, San Mateo, CA (1993) 177–183
293. Brock, W., Lakonishok, J. & LeBaron, B., : Simple technical trading rules and the stochastic properties of stock returns, Journal of Finance, 47, (1992), 1731–1764.
294. Robin, Franck, Andrea Orzati, Esteban Moreno, Otte Homan, and Werner Bachtold. : Simulation and evolutionary optimization of electron-beam lithography with genetic and simplex-downhill algorithms. IEEE Transactions on Evolutionary Computation, vol. 7, no. 1, p. 69–82 (February 2003).
295. Fraser, A. S. (1957) : Simulation of genetic systems by automatic digital computers. Australian Journal of Biological Science, 10, 484–491.
296. Langdon, W.B.: Size fair and homologous tree genetic programming crossovers. Genetic Programming and Evolvable Machines 1 (2000) 95–119.
297. Goldberg, D. E. (1989a) : Sizing populations for serial and parallel genetic algorithms.
298. Begley, Sharon and Gregory Beals. : Software au naturel. Newsweek, May 8, 1995, p. 70.
299. Lau, T.L. & Tsang, E.P.K., : Solving the processor configuration problem with a mutation-based genetic algorithm, International Journal on Artificial Intelligence Tools (IJAIT), World Scientific, Vol. 6, No. 4, (1997), 567–585.
300. Sweeney, R. J., : Some new filter rule test: Methods and results, Journal of Financial and Quantitative Analysis, 23, (1988), 285–300.
301. P. Turney, D. Whitley and R. Anderson, : Special issue on the baldwinian effect, Evolutionary Computation, vol. 4, no. 3, 1996, pp. 213–329.
302. Quarles, T., Newton, A.R., Pederson, D.O., and Sangiovanni-Vincentelli, A. (1994). : SPICE 3 Version 3F5 Users Manual. Department of Electrical Engineering and Computer Science, University of California. Berkeley, CA.
303. J. Schaffer, L. J. Eshelman, and D. Offutt, : Spurious correlations and premature convergence in genetic algorithms, in Foundations of Genetic Algorithms (G. J. E. Rawlings, Ed.), pp. 102–122, Morgan Kaufman, Los Altos, CA, 1991.
304. McKay, B., Willis, M. and Barton, G. (1997). : Steady state modelling of chemical process systems using genetic programming. Computers & Chemical Eng., Vol. 21, No. 9, pp. 981–986.
305. Lin G-H., Xue G., : Steiner problem with minimum number of Steiner points and bounded edge-length, Information Processing Letters, 69, 1999, 53–57.
306. Montana D.J, : Strongly typed genetic programming, Evolutionary computation, 3(2), 1995, pp. 199–230. Radcliffe N.J and Surry P.D, Formal memetic algorithms, Lecutre Notes in Computer Science 865, 1994.
307. Yang J and Soh C.K, : Structural optimization by genetic algorithms with tournament selection, Journal of Computing in Civil Engineering, July 1997, pp. 195–200.
308. Z. Fan, K. Seo, J. Hu, R. C. Rosenberg and E. D. Goodman, : System- Level Synthesis of MEMS via Genetic Programming and Bond Graphs, Genetic and Evolutionary Computation – GECCO-2003. Vol. 2724. 2003, pp. 2058–2071.
309. Shaefer, C. G. (1987) : The ARGOT strategy: adaptive representation genetic optimizer technique. Proceedings of the Second International Conference on Genetic Algorithms, 50–58. Cambridge, MA: Lawrence Erlbaum.
310. Meuleau, N. and Lattaud, C.(1995). : The artificial evolution of cooperation. In [Alliot et al., 1995], pages 159–180.
311. A. Teller, Evolving Programmers: The Co-evolution of Intelligent Recombination Operators, in Advances in Genetic Programming II, P. Angeline and K. Kinnear, editors. Cambridge: MIT Press., 1996.
312. S. G. Akl, : The Design and Analysis of Parallel Algorithms, Prentice-Hall, Englewood Cliffs, NJ, 1991.

313. Keane, A.J. and S.M. Brown. : The design of a satellite boom with enhanced vibration performance using genetic algorithm techniques. In Adaptive Computing in Engineering Design and Control 96 - Proceedings of the Second International Conference, I.C. Parmee (ed), p. 107–113. University of Plymouth, 1996.
314. Goldberg, D.E., : The Design of Innovation, Kluwer Academic Publishers, 2002
315. Belding T.C. (1995), : The Distributed Genetic Algorithm Revisited, Proceedings of the 6th Intl. Conf. on GAs, Morgan Kaufmann, pp. 122–129.
316. Campbell, J.Y., Lo, A.W. & MacKinlay, A.C., : The econometrics of financial markets, Princeton, N.J.: Princeton University Press, 1997.
317. Bird, Jon and Paul Layzell. : The evolved radio and its implications for modelling the evolution of novel sensors. In Proceedings of the 2002 Congress on Evolutionary Computation, p. 1836–1841.
318. Manderick, B., de Weger, M., & Spiessens, P. (1991) : The genetic algorithm and the structure of the fitness landscape. Proceedings of the Fourth International Conference on Genetic Algorithms, 143–149. La Jolla, CA: Morgan Kaufmann.
319. Koza, J.R. and Rice, J.P. (1992). Genetic Programming: The Movie. Cambridge, MA: MIT Press.
320. Schechter, Bruce. Putting a Darwinian spin on the diesel engine. : The New York Times, September 19, 2000, p. F3.
321. H. Muhlenbein, M. Schomish, and J. Born, : The parallel genetic algorithm as a function optimizer, Parallel Comput. 17 (1991), 619–632.
322. Atmar, W. (1992) : The philosophical errors that plague both evolutionary theory and simulated evolutionary programming. Proceedings of the First Annual Conference on Evolutionary Programming, 27–34. San Diego, CA: Evolutionary Programming Society.
323. Holland, J. (1986) Escaping brittleness : The possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In R. Michalski, J. Carbonell, T. Mitchell (eds.), Machine Learning: An Artificial Intelligence Approach. Los Altos: Morgan Kaufmann.
324. Howard D and Roberts SC, : The prediction of journey times on motorways using genetic programming, Cagnoni S et al (eds): EvoWorkshops 2002, LNCS 2279, Springer-Verlag, 2002, pp. 210–221.
325. Morgenstern, L.: The problem with solutions to the frame problem. In Ford, K.M., Pylyshyn, Z., eds.: The Robots Dilemma Revisited: The Frame Problem in Artificial Intelligence. Ablex Publishing Co., Norwood, New Jersey (1996) 99–133
326. J. M. Link, et al., : The target silicon detector for the FOCUS spectrometer, Nucl. Instrum. Meth. A516 (2003) 364–376.
327. Fudenberg, D. and Levine, D. K. (1998). : The Theory of Learning in Games. MIT Press.
328. Petit, Charles. : Touched by nature: Putting evolution to work on the assembly line. U.S. News and World Report, vol. 125, no. 4, p. 43–45 (July 27, 1998). Available online at http://www.genetic-programming.com/published/usnwr072798.html.
329. Chen, S-H. & Yeh, C-H., : Toward a computable approach to the efficient market hypothesis: An application of genetic programming, Journal of Economic Dynamics and Control, 21, (1996), 1043–1063.
330. X. Yao and Y. Liu, : Toward designing artificial neural networks by evolution, Appl. Math. Computation, vol. 91, no. 1, 1998, pp. 83–90.
331. Harp, S. A., Samad, T., & Guha, A. (1991) : Towards the genetic synthesis of neural networks. Proceedings of the Fourth International Conference on Genetic Algorithms, 360–369. La Jolla, CA: Morgan Kaufmann.
332. G. W. Greenwood, : Training partially recurrent neural networks using evolutionary strategies, IEEE Trans. Speech Audio Processing, vol. 5, 1997, pp. 192–194.
333. D. J. Janson and J. F. Frenzel, : Training product unit neural networks with genetic algorithms, IEEE Expert, vol. 8, 1993, pp. 26–33.
334. Durst, S. (1992) : Tree annealing vs. Gradient Descent Methods. Term project for Numerical Optimization course, Dept. of Electrical Engineering, Carnegie Mellon University.

335. R. S. Sutton, : Two problems with backpropagation and other steepestdescent learning procedure for networks, Proc. 8th Annual Conf. Cognitive Science Society, Hillsdale, NJ: Erlbaum, 1986, pp. 823–831.
336. Alba, E., Cotta, C. and Troya, J.M. (1996). : Typeconstrained Genetic Programming for Rule-Base Definition in Fuzzy Logic Controllers, GP 96- Proceedings of the 1st Annual Conference, July 28–31, pp. 255–260
337. C. J. Mertz and P. M. Murphy, : UCI repository of machine learning databases. http://www-old.ics.uci.edu/pub/machine-learning-databases, 2002
338. Hughes, Evan and Maurice Leyland. : Using multiple genetic algorithms to generate radar point-scatterer models. IEEE Transactions on Evolutionary Computation, vol. 4, no. 2, p. 147–163 (July 2000).
339. K. A. De Jong, M. A. Potter, and W. M. Spears, : Using problem generators to explore the effects of epistasis, in Proceedings of the 7th International Conference of Genetic Algorithms (T. Back, Ed.), pp. 338–345, Morgan Kaufman, Los Altos, CA, 1997.
340. Fujiko, C., & Dickinson, J. (1987) : Using the genetic algorithm to generate LISP source code to solve the prisoners dilemma. Proceedings of the Second International Conference on Genetic Algorithms, 236–240. Cambridge, MA: Lawrence Erlbaum.
341. Chellapilla, Kumar and David Fogel. : Verifying Anacondas expert rating by competing against Chinook: experiments in co-evolving a neural checkers player. Neurocomputing, vol. 42, no. 1–4, p. 69–86 (January 2002).
342. Diada J.M et al, : Visualizing tree structures in genetic programming, Lecture Notes in Computer Science 2724, 2003, pp. 1652–1664.
343. Lo, A.W. & MacKinlay, A.C., : When are contrarian profits due to stock market overreaction? Review of Financial Studies 3, (1990), 175–206.
344. Dembski, William. No Free Lunch : Why Specified Complexity Cannot Be Purchased Without Intelligence. Rowman & Littlefield, 2002.
345. Dawkins, Richard. The Blind Watchmaker : Why the Evidence of Evolution Reveals a Universe Without Design. W.W. Norton, 1996.